Desarrollo de Videojuegos

Un Enfoque Práctico

Escuela Superior de Informática
Universidad de Castilla-La Mancha

D. Vallejo · C. González · D. Villa · F. Jurado
F. Moya · J. A. Albusac · C. Martín · S. Pérez
F. J. Villanueva · C. Mora · J. J. Castro
M. A. Redondo · L. Jiménez · J. López · M. García
M. Palomo · G. Simmross · J. L. González

Desarrollo de Videojuegos

Un Enfoque Práctico

Desarrollo de Videojuegos

Un Enfoque Práctico

Escuela Superior de Informática
Universidad de Castilla-La Mancha

D. Vallejo · C. González · D. Villa · F. Jurado
F. Moya · J. A. Albusac · C. Martín · S. Pérez
F. J. Villanueva · C. Mora · J. J. Castro
M. A. Redondo · L. Jiménez · J. López · M. García
M. Palomo · G. Simmross · J. L. González

Título: Desarrollo de Videojuegos: Un Enfoque Práctico
Edición: Julio 2014
Autores: David Vallejo Fernández, Carlos González Morcillo, David Villa Alises, Francisco Jurado Monroy, Francisco Moya Fernández, Javier Alonso Albusac Jiménez, Cleto Martín Angelina, Sergio Pérez Camacho, Félix Jesús Villanueva Molina, César Mora Castro, José Jesús Castro Sánchez, Miguel Ángel Redondo Duque, Luis Jiménez Linares, Jorge López González, Miguel García Corchero, Manuel Palomo Duarte, Guillermo Simmross Wattenberg, José Luis González Sánchez. *Colaboradores:* David Frutos Talavera, Sergio Fernández Durán
ISBN: 978-84-942382-9-1 **Depósito Legal:** VG 415-2014
Publica: EdLibrix
Edita: David Vallejo, Carlos González y David Villa
Portada: (Ilustración de portada) Víctor Barba Pizarro
Diseño: Carlos González Morcillo y Víctor Barba Pizarro

Este libro fue compuesto con LaTeX a partir de una plantilla de David Villa Alises y Carlos González Morcillo.

Escuela Superior de Informática
Departamento de Tecnología y Sistemas de Información
Paseo de la Universidad, 4
13071, Ciudad Real
Email: esi@uclm.es

Creative Commons License: Usted es libre de copiar, distribuir y comunicar públicamente la obra, bajo las condiciones siguientes: 1. Reconocimiento. Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador. 2. No comercial. No puede utilizar esta obra para fines comerciales. 3. Sin obras derivadas. No se puede alterar, transformar o generar una obra derivada a partir de esta obra. Más información en: http://creativecommons.org/licenses/by-nc-nd/3.0/

Prefacio

La industria del videojuego ocupa el primer lugar en el ocio audio-visual e interactivo a nivel mundial, por encima de industrias tan potentes como el cine o la música. Como consecuencia directa, existe una gran demanda de profesionales cualificados para diseñar y desarrollar videojuegos no sólo para consolas y ordenadores, sino también para el más que creciente mercado de los teléfonos móviles.

Este libro, dividido en cuatro bloques, tiene como objetivo principal proporcionar los conocimientos necesarios para llevar a cabo dicha tarea desde una perspectiva esencialmente técnica:

1. **Arquitectura del Motor**, donde se estudian los aspectos esenciales del diseño de un motor de videojuegos, así como las técnicas básicas de programación y patrones de diseño. En este bloque también se estudian los conceptos más relevantes del lenguaje de programación C++.

2. **Programación Gráfica**, donde se presta especial atención a los algoritmos y técnicas de representación gráfica, junto con las optimizaciones en sistemas de despliegue interactivo.

3. **Técnicas Avanzadas**, donde se recogen ciertos aspectos avanzados, como estructuras de datos específicas, técnicas de validación y pruebas o simulación física. Así mismo, en este bloque se profundiza en el lenguaje C++.

4. **Desarrollo de Componentes**, donde, finalmente, se detallan ciertos componentes específicos del motor, como la Inteligencia Artificial, Networking, Sonido y Multimedia o técnicas avanzadas de Interacción.

Sobre este libro

Este libro que tienes en tus manos es una ampliación y revisión de los apuntes del *Curso de Experto en Desarrollo de Videojuegos*, impartido en la Escuela Superior de Informática de Ciudad Real de la Universidad de Castilla-La Mancha. Puedes obtener más información sobre el curso, así como los resultados de los trabajos creados por los alumnos, en la web del mismo: http://www.cedv.es. La versión electrónica de este libro puede descargarse desde la web anterior. El libro «físico» puede adquirirse desde la página web de la editorial online *Edlibrix* en http://www.shoplibrix.com.

Requisitos previos

Este libro tiene un público objetivo con un perfil principalmente técnico. Al igual que el curso del que surgió, está orientado a la capacitación de profesionales de la programación de videojuegos. De esta forma, este libro no está orientado para un público de perfil artístico (modeladores, animadores, músicos, etc.) en el ámbito de los videojuegos.

Se asume que el lector es capaz de desarrollar programas de nivel medio en C y C++. Aunque se describen algunos aspectos clave de C++ a modo de resumen, es recomendable refrescar los conceptos básicos con alguno de los libros recogidos en la bibliografía. De igual modo, se asume que el lector tiene conocimientos de estructuras de datos y algoritmia. El libro está orientado principalmente para titulados o estudiantes de últimos cursos de Ingeniería en Informática.

Programas y código fuente

El código de los ejemplos del libro pueden descargarse en la siguiente página web: http://www.cedv.es. Salvo que se especifique explícitamente otra licencia, todos los ejemplos del libro se distribuyen bajo GPLv3.

Agradecimientos

Los autores del libro quieren agradecer, en primer lugar, a los alumnos de las tres primeras ediciones del *Curso de Experto en Desarrollo de Videojuegos* por su participación en el mismo y el excelente ambiente en las clases, las cuestiones planteadas y la pasión demostrada en el desarrollo de todos los trabajos.

De igual modo, se quiere reflejar el agradecimiento especial al personal de administración y servicios de la Escuela Superior de Informática, por su soporte, predisposición y ayuda en todos los caprichosos requisitos que planteábamos a lo largo del curso.

Por otra parte, este agradecimiento también se hace extensivo a la Escuela de Informatica de Ciudad Real y al Departamento de Tecnologías y Sistema de Información de la Universidad de Castilla-La Mancha.

Finalmente, los autores desean agradecer su participación a los colaboradores de las tres primeras ediciones: *Indra Software Labs*, la asociación de desarrolladores de videojuegos *Stratos*, *Libro Virtual*, *Devilish Games*, *Dolores Entertainment*, *From the Bench*, *Iberlynx*, *KitMaker*, *Playspace*, *Totemcat/Materia Works* y *ZuinqStudio*.

Autores

David Vallejo (2009, Doctor Europeo en Informática, Universidad de Castilla-La Mancha) es Profesor Ayudante Doctor e imparte docencia en la Escuela de Informática de Ciudad Real (UCLM) en asignaturas relacionadas con Informática Gráfica, Programación y Sistemas Operativos desde 2007. Actualmente, su actividad investigadora gira en torno a la Vigilancia Inteligente, los Sistemas Multi-Agente y el Rendering Distribuido.

Carlos González (2007, Doctor Europeo en Informática, Universidad de Castilla-La Mancha) es Profesor Titular de Universidad e imparte docencia en la Escuela de Informática de Ciudad Real (UCLM) en asignaturas relacionadas con Informática Gráfica, Síntesis de Imagen Realista y Sistemas Operativos desde 2002. Actualmente, su actividad investigadora gira en torno a los Sistemas Multi-Agente, el Rendering Distribuido y la Realidad Aumentada.

David Villa (2009, Doctor Ingeniero Informático, Universidad de Castilla-La Mancha) es Profesor Ayudante Doctor e imparte docencia en la Escuela de Informática de Ciudad Real (UCLM) en materias relacionadas con las redes de computadores y sistemas distribuidos desde el 2002. Sus intereses profesionales se centran en los sistemas empotrados en red, los sistemas ubicuos y las redes heterogéneas y virtuales. Es experto en métodos de desarrollo ágiles y en los lenguajes C++ y Python. Colabora con el proyecto Debian como maintainer de paquetes oficiales.

Francisco Jurado (2010, Doctor Europeo en Informática, Universidad de Castilla-La Mancha) es Profesor Ayudante Doctor en la Universidad Autónoma de Madrid. Su actividad investigadora actual gira en torno a la aplicación de técnicas de Ingeniería del Software e Inteligencia Artificial al ámbito del eLearning, los Sistemas Tutores, los Sistemas Adaptativos y los Entornos Colaborativos.

Francisco Moya (2003, Doctor Ingeniero en Telecomunicación, Universidad Politécnica de Madrid). Desde 1999 trabaja como profesor de la Escuela Superior de Informática de la Universidad de Castilla la Mancha, desde 2008 como Profesor Contratado Doctor. Sus actuales líneas de investigación incluyen los sistemas distribuidos heterogéneos, la automatización del diseño electrónico y sus aplicaciones en la construcción de servicios a gran escala y en el diseño de sistemas en chip. Desde 2007 es también Debian Developer.

Javier Albusac (2009, Doctor Europeo en Informática, Universidad de Castilla-La Mancha) es Profesor Ayudante Doctor e imparte docencia en la Escuela de Ingeniería Minera e Industrial de Almadén (EIMIA) en las asignaturas de Informática, Ofimática Aplicada a la Ingeniería y Sistemas de Comunicación en Edificios desde 2007. Actualmente, su actividad investigadora gira en torno a la Vigilancia Inteligente, Robótica Móvil y Aprendizaje Automático.

Cleto Martín (2011, Ingeniero Informática y Máster de Investigación en Tecnologías Informáticas Avanzadas, Universidad de Castilla-La Mancha) trabaja como Software Developer en Digital TV Labs (Bristol, UK) y como mantenedor de paquetes de aplicaciones para Canonical Ltd. y el proyecto Debian. Es un gran entusiasta de los sistemas basados en GNU/Linux, así como el desarrollo de aplicaciones basadas en redes de computadores y sistemas distribuidos.

Sergio Pérez (2011, Ingeniero en Informática, Universidad de Castilla-La Mancha) trabaja como ingeniero consultor diseñando software de redes para Ericsson R&D. Sus intereses principales son GNU/Linux, las redes, los videojuegos y la realidad aumentada.

Félix J. Villanueva (2009, Doctor en Ingeniería Informática, Universidad de Castilla-La Mancha) es contratado doctor e imparte docencia en el área de tecnología y arquitectura de computadores. Las asignaturas que imparte se centran en el campo de las redes de computadores con una experiencia docente de más de diez años. Sus principales campos de investigación en la actualidad son redes inalámbricas de sensores, entornos inteligentes y sistemas empotrados.

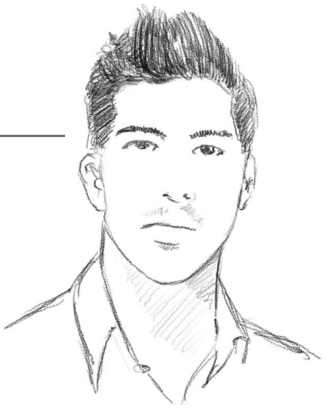
César Mora (2013, Master en Computer Science por la Universidad de Minnesota, 2011 Ingeniero en Informática, Universidad de Casilla-La Mancha). Sus temas de interés están relacionados con la Informática Gráfica, la Visión Artificial y la Realidad Aumentada.

José Jesús Castro (2001, Doctor en Informática, Universidad de Granada) es Profesor Titular de Universidad en el área de Lenguajes y Sistemas Informáticos, desde 1999 imparte docencia en la Escuela Superior de Informática de la UCLM. Sus temas de investigación están relacionados con el uso y desarrollo de métodos de IA para la resolución de problemas reales, donde cuenta con una amplia experiencia en proyectos de investigación, siendo autor de numerosas publicaciones.

Miguel Ángel Redondo (2002, Doctor en Ingeniería Informática, Universidad de Castilla – La Mancha) es Profesor Titular de Universidad en la Escuela Superior de Informática de la UCLM en Ciudad Real, impartiendo docencia en asignaturas relacionadas con Interacción Persona-Computador y Sistemas Operativos. Su actividad investigadora se centra en la innovación y aplicación de técnicas de Ingeniería del Software al desarrollo de sistemas avanzados de Interacción Persona-Computador y al desarrollo de sistemas de e-Learning.

Luis Jiménez (1997, Doctor en Informática, Universidad de Granada) es Titular de Universidad e imparte docencia en la Escuela de Informática de Ciudad Real (UCLM) en asignaturas relacionadas la Inteligencia Artificial y Softcomputing desde 1995. Actualmente, su actividad investigadora gira en torno a los Sistemas Inteligentes aplicados mediante Sistemas Multi-Agente, técnicas de softcomputing e inteligencia artificial distribuida.

Jorge López (2011, Ingeniero en Informática por la UCLM y Máster en Diseño y Desarrollo de videojuegos por la UCM). Especializado en desarrollo 3D con C++ y OpenGL, y en el engine Unity 3D. Actualmente trabaja como programador en Totemcat – Materia Works.



Miguel García es desarrollador independiente de Videojuegos en plataformas iOS, Android, Mac OS X, GNU/Linux y MS Windows y socio fundador de Atomic Flavor. Actualmente dirige el estudio de desarrollo de videojuegos independientes Quaternion Studio.

Manuel Palomo (2011, Doctor por la Universidad de Cádiz) es Profesor Contratado Doctor e imparte docencia en la Escuela Superior de Ingeniería de la Universidad de Cádiz en asignaturas relacionadas con el Diseño de Videojuegos, Recuperación de la Información y Sistemas Informáticos Abiertos. Actualmente su actividad investigadora se centra en las tecnologías del aprendizaje, principalmente videojuegos educativos y los sistemas colaborativos de desarrollo y documentación.

Guillermo Simmross (2003, Ingeniero Técnico de Telecomunicación, 2005 Ingeniero en Electrónica y 2008, Máster Dirección de Proyectos, Universidad de Valladolid) es Compositor y diseñador de sonido freelance e imparte docencia en colaboración con la Universidad Camilo José Cela sobre Composición de Música para Videojuegos. Actualmente trabaja como responsable de producto en Optimyth Software.

José Luis González (2010, Doctor en Informática, Universidad de Granada). Especialista en calidad y experiencia de usuario en sistemas interactivos y videojuegos, temas donde imparte su docencia e investiga. Ha colaborado con distintas compañías del sector, como Nintendo o MercurySteam. Es autor de distintos libros sobre la jugabilidad y el diseño y evaluación de la experiencia del jugador.

Índice general

I Arquitectura del Motor **1**

1. Introducción **3**

1.1. El desarrollo de videojuegos 3

1.1.1. La industria del videojuego. Presente y futuro 3

1.1.2. Estructura típica de un equipo de desarrollo 5

1.1.3. El concepto de juego 7

1.1.4. Motor de juego 9

1.1.5. Géneros de juegos 10

1.2. Arquitectura del motor. Visión general 15

1.2.1. Hardware, *drivers* y sistema operativo 15

1.2.2. SDKs y *middlewares* 16

1.2.3. Capa independiente de la plataforma 17

1.2.4. Subsistemas principales 18

1.2.5. Gestor de recursos 18

1.2.6. Motor de *rendering* 19

1.2.7. Herramientas de depuración 22

1.2.8. Motor de física 22

1.2.9. Interfaces de usuario 23

1.2.10. *Networking* y multijugador 23

1.2.11. Subsistema de juego 24

1.2.12. Audio 26

1.2.13. Subsistemas específicos de juego 26

2. Herramientas de Desarrollo **27**

2.1. Introducción 27
2.2. Compilación, enlazado y depuración 28
2.2.1. Conceptos básicos 28
2.2.2. Compilando con GCC 31
2.2.3. ¿Cómo funciona GCC? 31
2.2.4. Ejemplos 33
2.2.5. Otras herramientas 39
2.2.6. Depurando con GDB 39
2.2.7. Construcción automática con GNU Make 45
2.3. Gestión de proyectos y documentación 50
2.3.1. Sistemas de control de versiones 50
2.3.2. Documentación 59
2.3.3. Forjas de desarrollo 62

3. C++. Aspectos Esenciales 65
3.1. Utilidades básicas 65
3.1.1. Introducción a C++ 65
3.1.2. ¡Hola Mundo! en C++ 66
3.1.3. Tipos, declaraciones y modificadores 67
3.1.4. Punteros, arrays y estructuras 68
3.1.5. Referencias y funciones 71
3.2. Clases 74
3.2.1. Fundamentos básicos 74
3.2.2. Aspectos específicos de las clases 77
3.2.3. Sobrecarga de operadores 81
3.3. Herencia y polimorfismo 84
3.3.1. Herencia 84
3.3.2. Herencia múltiple 86
3.3.3. Funciones virtuales y polimorfismo 90
3.4. Plantillas 94
3.4.1. Caso de estudio. Listas 94
3.4.2. Utilizando plantillas en C++ 96
3.4.3. ¿Cuándo utilizar plantillas? 98
3.5. Manejo de excepciones 99
3.5.1. Alternativas existentes 99
3.5.2. Excepciones en C++ 100
3.5.3. ¿Cómo manejar excepciones adecuadamente? 103
3.5.4. ¿Cuándo utilizar excepciones? 105

4. Patrones de Diseño **107**
4.1. Introducción . . . 108
4.1.1. Estructura de un patrón de diseño . . . 109
4.1.2. Tipos de patrones . . . 110
4.2. Patrones de creación . . . 110
4.2.1. Singleton . . . 110
4.2.2. Abstract Factory . . . 112
4.2.3. Factory Method . . . 115
4.2.4. Prototype . . . 116
4.3. Patrones estructurales . . . 117
4.3.1. Composite . . . 118
4.3.2. Decorator . . . 119
4.3.3. Facade . . . 121
4.3.4. MVC . . . 123
4.3.5. Adapter . . . 124
4.3.6. Proxy . . . 125
4.4. Patrones de comportamiento . . . 127
4.4.1. Observer . . . 128
4.4.2. State . . . 130
4.4.3. Iterator . . . 131
4.4.4. Template Method . . . 133
4.4.5. Strategy . . . 135
4.4.6. Reactor . . . 136
4.4.7. Visitor . . . 138
4.5. *Programming Idioms* . . . 141
4.5.1. Orthodox Canonical Form . . . 141
4.5.2. Interface Class . . . 142
4.5.3. Final Class . . . 143
4.5.4. pImpl . . . 144

5. La Biblioteca STL **147**
5.1. Visión general de STL . . . 147
5.2. STL y el desarrollo de videojuegos . . . 151
5.2.1. Reutilización de código . . . 151
5.2.2. Rendimiento . . . 152
5.2.3. Inconvenientes . . . 152
5.3. Secuencias . . . 153
5.3.1. Vector . . . 153
5.3.2. Deque . . . 156

5.3.3. List 158
5.4. Contenedores asociativos 161
5.4.1. Set y multiset 161
5.4.2. Map y multimap 164
5.5. Adaptadores de secuencia 167
5.5.1. Stack 167
5.5.2. Queue 168
5.5.3. Cola de prioridad 168

6. Gestión de Recursos **171**
6.1. El bucle de juego 172
6.1.1. El bucle de renderizado 172
6.1.2. Visión general del bucle de juego 172
6.1.3. Arquitecturas típicas del bucle de juego 174
6.1.4. Gestión de estados de juego con Ogre3D 177
6.1.5. Definición de estados concretos 182
6.2. Gestión básica de recursos 184
6.2.1. Gestión de recursos con Ogre3D 184
6.2.2. Gestión básica del sonido 186
6.3. El sistema de archivos 195
6.3.1. Gestión y tratamiento de archivos 196
6.3.2. E/S básica 198
6.3.3. E/S asíncrona 200
6.3.4. Caso de estudio. La biblioteca Boost.Asio C++ 203
6.3.5. Consideraciones finales 207
6.4. Importador de datos de intercambio 207
6.4.1. Formatos de intercambio 208
6.4.2. Creación de un importador 210

7. Bajo Nivel y Concurrencia **219**
7.1. Subsistema de arranque y parada 220
7.1.1. Aspectos fundamentales 220
7.1.2. Esquema típico de arranque y parada 223
7.1.3. Caso de estudio. Ogre 3D 224
7.1.4. Caso de estudio. Quake III 227
7.2. Contenedores 229
7.2.1. Iteradores 230
7.2.2. Más allá de STL 231
7.3. Subsistema de gestión de cadenas 235

7.3.1. Cuestiones específicas . . . 235
7.3.2. Optimizando el tratamiento de cadenas . . . 236
7.3.3. *Hashing* de cadenas . . . 238
7.4. Configuración del motor . . . 239
7.4.1. Esquemas típicos de configuración . . . 239
7.4.2. Caso de estudio. Esquemas de definición. . . . 240
7.5. Fundamentos básicos de concurrencia . . . 241
7.5.1. El concepto de hilo . . . 241
7.5.2. El problema de la sección crítica . . . 242
7.6. La biblioteca de hilos de ICE . . . 244
7.6.1. Internet Communication Engine . . . 244
7.6.2. Manejo de hilos . . . 245
7.6.3. Exclusión mutua básica . . . 247
7.6.4. Flexibilizando el concepto de *mutex* . . . 251
7.6.5. Introduciendo monitores . . . 251
7.7. Concurrencia en C++11 . . . 256
7.7.1. Filósofos comensales en C++11 . . . 258
7.8. *Multi-threading* en Ogre3D . . . 260
7.9. Caso de estudio. Procesamiento en segundo plano mediante hilos . . . 263

II Programación Gráfica 267

8. Fundamentos de Gráficos Tridimensionales 269
8.1. Introducción . . . 269
8.2. El Pipeline Gráfico . . . 272
8.2.1. Etapa de Aplicación . . . 272
8.2.2. Etapa de Geometría . . . 272
8.2.3. Etapa Rasterización . . . 275
8.2.4. Proyección en Perspectiva . . . 276
8.3. Implementación del Pipeline en GPU . . . 277
8.3.1. Vertex Shader . . . 278
8.3.2. Geometry Shader . . . 279
8.3.3. Pixel Shader . . . 279
8.4. Arquitectura del motor gráfico . . . 279
8.5. Casos de Estudio . . . 280
8.6. Introducción a OGRE . . . 281
8.6.1. Arquitectura General . . . 284
8.6.2. Instalación . . . 288

8.7. Hola Mundo en OGRE 289

9. Matemáticas para Videojuegos **295**

9.1. Puntos, Vectores y Coordenadas 295

9.1.1. Sistemas de Referencia 295

9.1.2. Puntos 296

9.1.3. Vectores 297

9.2. Transformaciones Geométricas 300

9.2.1. Representación Matricial 301

9.2.2. Transformaciones Inversas 303

9.2.3. Composición 303

9.3. Perspectiva: Representación Matricial 304

9.4. Cuaternios 306

9.4.1. Suma y Multiplicación 308

9.4.2. Inversa 308

9.4.3. Rotación empleando Cuaternios 308

9.5. Interpolación Lineal y Esférica 309

9.6. El Módulo Math en OGRE 310

9.7. Ejercicios Propuestos 311

10. Grafos de Escena **313**

10.1. Justificación 313

10.1.1. Operaciones a Nivel de Nodo 314

10.2. El Gestor de Escenas de OGRE 315

10.2.1. Creación de Objetos 316

10.2.2. Transformaciones 3D 317

10.2.3. Espacios de transformación 319

11. Recursos Gráficos y Sistema de Archivos **321**

11.1. Formatos de Especificación 321

11.1.1. Introducción 321

11.1.2. Recursos de gráficos 3D: formatos y requerimientos de almacenamiento 323

11.1.3. Casos de Estudio 327

11.2. Exportación y Adaptación de Contenidos 338

11.2.1. Instalación del exportador de Ogre en Blender 338

11.2.2. Creación de un modelo en Blender 340

11.2.3. Aplicación de texturas mediante UV Mapping 340

11.2.4. Exportación del objeto en formato Ogre XML 343

11.2.5. Carga del objeto en una aplicación Ogre 345

11.3. Procesamiento de Recursos Gráficos 346
11.3.1. Ejemplo de uso 348
11.4. Gestión de Recursos y Escena 351
11.4.1. Recursos empaquetados 352
11.4.2. Gestión del ratón 352
11.4.3. Geometría Estática 353
11.4.4. Queries 354
11.4.5. Máscaras 357

12. APIS de Gráficos 3D **359**
12.1. Introducción 359
12.2. Modelo Conceptual 360
12.2.1. Cambio de Estado 361
12.2.2. Dibujar Primitivas 362
12.3. Pipeline de OpenGL 362
12.3.1. Transformación de Visualización 363
12.3.2. Transformación de Modelado 363
12.3.3. Transformación de Proyección 364
12.3.4. Matrices 364
12.3.5. Dos ejemplos de transformaciones jerárquicas 366
12.4. Ejercicios Propuestos 368

13. Gestión Manual OGRE 3D **369**
13.1. Inicialización Manual 369
13.1.1. Inicialización 370
13.1.2. Carga de Recursos 373
13.1.3. FrameListener 373
13.2. Uso de OIS 374
13.2.1. Uso de Teclado y Ratón 376
13.3. Creación manual de Entidades 377
13.4. Uso de Overlays 379

14. Interacción y Widgets **383**
14.1. Interfaces de usuario en videojuegos 383
14.1.1. Menú 386
14.1.2. HUD 386
14.2. Introducción CEGUI 387
14.2.1. Instalación 388
14.2.2. Inicialización 388

14.2.3. El Sistema de Dimensión Unificado 390
14.2.4. Detección de eventos de entrada 391
14.3. Primera aplicación 393
14.4. Tipos de *Widgets* 395
14.5. *Layouts* 396
14.6. Ejemplo de interfaz 397
14.7. Editores de *layouts* gráficos 400
14.8. Scripts en detalle 400
14.8.1. *Scheme* 400
14.8.2. *Font* 401
14.8.3. *Imageset* 401
14.8.4. *LookNFeel* 402
14.9. Cámara de Ogre en un *Window* 402
14.10Formateo de texto 404
14.10.1Introducción 404
14.10.2Color 405
14.10.3Formato 405
14.10.4Insertar imágenes 405
14.10.5Alineamiento vertical 406
14.10.6*Padding* 406
14.10.7Ejemplo de texto formateado 407
14.11Características avanzadas 408

15. Materiales y Texturas **409**
15.1. Introducción 409
15.2. Modelos de Sombreado 410
15.3. Mapeado de Texturas 412
15.4. Materiales en Ogre 413
15.4.1. Composición 414
15.4.2. Ejemplo de Materiales 415
15.5. Mapeado UV en Blender 417
15.5.1. Costuras 420
15.6. Ejemplos de Materiales en Ogre 421
15.7. Render a Textura 424
15.7.1. Texture Listener 426
15.7.2. Espejo (Mirror) 427

16. Iluminación **429**
16.1. Introducción 429

16.2. Tipos de Fuentes de Luz 430
16.3. Sombras Estáticas Vs Dinámicas 431
16.3.1. Sombras basadas en Stencil Buffer 432
16.3.2. Sombras basadas en Texturas 434
16.4. Ejemplo de uso 435
16.5. Mapas de Iluminación 437
16.6. Ambient Occlusion 439
16.7. Radiosidad 440

17. Exportación y Uso de Datos de Intercambio **445**
17.1. Introducción 445

18. Animación **451**
18.1. Introducción 451
18.1.1. Animación Básica 452
18.1.2. Animación de Alto Nivel 453
18.2. Animación en Ogre 454
18.2.1. Animación Keyframe 454
18.2.2. Controladores 456
18.3. Exportación desde Blender 456
18.4. Mezclado de animaciones 459

19. Simulación Física **465**
19.1. Introducción 465
19.1.1. Algunos Motores de Simulación 466
19.1.2. Aspectos destacables 467
19.1.3. Conceptos Básicos 468
19.2. Sistema de Detección de Colisiones 469
19.2.1. Formas de Colisión 470
19.2.2. Optimizaciones 472
19.2.3. Preguntando al sistema... 472
19.3. Dinámica del Cuerpo Rígido 473
19.4. Restricciones 474
19.5. Introducción a Bullet 475
19.5.1. Pipeline de Físicas de Cuerpo Rígido 476
19.5.2. Hola Mundo en Bullet 477
19.6. Integración manual en Ogre 484
19.7. Hola Mundo en OgreBullet 487
19.8. RayQueries 491

19.9. TriangleMeshCollisionShape 493
19.10Detección de colisiones 494
19.11Restricción de Vehículo 496
19.12Determinismo 500
19.13Escala de los Objetos 502
19.14Serialización 502

III Técnicas Avanzadas de Desarrollo 505

20. Aspectos de Jugabilidad y Metodologías de Desarrollo 507
20.1. Jugabilidad y Experiencia del Jugador 507
20.1.1. Introducción 507
20.1.2. Caracterización de la Jugabilidad 508
20.1.3. Facetas de la Jugabilidad 510
20.1.4. Calidad de un juego en base a la Jugabilidad 512
20.2. Metodologías de Producción y Desarrollo 518
20.2.1. Pre-Producción 519
20.2.2. Producción 521
20.2.3. Post-Producción 523
20.3. Metodologías Alternativas 524
20.3.1. Proceso Unificado del Juego 524
20.3.2. Desarrollo Incremental 524
20.3.3. Desarrollo Ágil y Scrum 524
20.3.4. Desarrollo Centrado en el Jugador 525

21. C++ Avanzado 527
21.1. Estructuras de datos no lineales 527
21.1.1. Árboles binarios 528
21.1.2. Recorrido de árboles 542
21.1.3. *Quadtree* y *octree* 545
21.2. Patrones de diseño avanzados 548
21.2.1. Smart pointers 548
21.2.2. Command 554
21.2.3. Curiously recurring template pattern 557
21.2.4. Reactor 561
21.2.5. Acceptor/Connector 564
21.3. Programación genérica 568
21.3.1. Algoritmos 568

21.3.2. Predicados . . . 571
21.3.3. Functors . . . 572
21.3.4. Adaptadores . . . 574
21.3.5. Algoritmos idempotentes . . . 576
21.3.6. Algoritmos de transformación . . . 578
21.3.7. Algoritmos de ordenación . . . 582
21.3.8. Algoritmos numéricos . . . 584
21.3.9. Ejemplo: inventario de armas . . . 585
21.4. Aspectos avanzados de la STL . . . 588
21.4.1. Eficiencia . . . 588
21.4.2. Semántica de copia . . . 591
21.4.3. Extendiendo la STL . . . 593
21.4.4. Allocators . . . 596
21.4.5. Novedades de la STL en C++11 . . . 599
21.5. C++11: Novedades del nuevo estándar . . . 603
21.5.1. Compilando con `g++` y `clang` . . . 603
21.5.2. Cambios en el núcleo del lenguaje . . . 603
21.5.3. Cambios en la biblioteca de C++ . . . 616
21.6. Plugins . . . 619
21.6.1. Entendiendo las bibliotecas dinámicas . . . 620
21.6.2. Plugins con `libdl` . . . 622
21.6.3. Plugins con Glib `gmodule` . . . 627
21.6.4. Carga dinámica desde Python . . . 629
21.6.5. Plugins como objetos mediante el patrón *Factory Method* . . . 629
21.6.6. Plugins multi-plataforma . . . 632

22. Técnicas específicas **635**
22.1. Serialización de objetos . . . 635
22.1.1. *Streams* . . . 636
22.1.2. Serialización y Dependencias entre objetos . . . 639
22.1.3. Serialización con Boost . . . 649
22.2. C++ y scripting . . . 653
22.2.1. Consideraciones de diseño . . . 653
22.2.2. Invocando Python desde C++ de forma nativa . . . 655
22.2.3. Librería boost . . . 656
22.2.4. Herramienta SWIG . . . 660
22.2.5. Conclusiones . . . 662

23. Optimización **663**

23.1. Perfilado de programas 664
23.1.1. El perfilador de Linux *perf* 666
23.1.2. Obteniendo ayuda 668
23.1.3. Estadísticas y registro de eventos 668
23.1.4. Multiplexación y escalado 669
23.1.5. Métricas por hilo, por proceso o por CPU 670
23.1.6. Muestreo de eventos 671
23.1.7. Otras opciones de `perf` 675
23.1.8. Otros perfiladores 675
23.2. Optimizaciones del compilador 677
23.2.1. Variables registro 677
23.2.2. Código estático y funciones *inline* 678
23.2.3. Eliminación de copias 683
23.2.4. Volatile 684
23.3. Conclusiones 685

24. Validación y Pruebas **687**
24.1. Programación defensiva 687
24.1.1. Sobrecarga 689
24.2. Desarrollo ágil 690
24.3. TDD 690
24.3.1. Las pruebas primero 691
24.3.2. rojo, verde, refactorizar 691
24.4. Tipos de pruebas 692
24.5. Pruebas unitarias con google-tests 693
24.6. Dobles de prueba 695
24.7. Dobles de prueba con google-mock 696
24.8. Limitaciones 701

25. Empaquetado y distribución **703**
25.1. Empaquetado y distribución en Windows 704
25.1.1. Creación de un paquete básico 704
25.1.2. Interacción con el usuario 710
25.1.3. Otras características 711
25.2. Empaquetado y distribución en GNU/Linux 712
25.2.1. Pidiendo un paquete 713
25.2.2. Obteniendo el fuente original 714
25.2.3. Estructura básica 714
25.2.4. Construcción del paquete 719

25.2.5. Parches: adaptación a Debian 721
25.2.6. Actualización del paquete 724
25.2.7. Subir un paquete a Debian 725
25.3. Otros formatos de paquete 725

26. Representación Avanzada **727**
26.1. Fundamentos 727
26.1.1. Billboards 727
26.1.2. Sistemas de partículas 730
26.2. Uso de Billboards 731
26.2.1. Tipos de Billboard 733
26.2.2. Aplicando texturas 733
26.3. Uso de Sistemas de Partículas 734
26.3.1. Emisores 735
26.3.2. Efectores 736
26.3.3. Ejemplos de Sistemas de Partículas 737
26.4. Introducción a los Shaders 739
26.4.1. Un poco de historia 739
26.4.2. ¿Y qué es un Shader? 741
26.4.3. Pipelines Gráficos 741
26.4.4. Fixed-Function Pipeline 742
26.4.5. Programmable-Function Pipeline 745
26.4.6. Aplicaciones de los Shader 748
26.4.7. Lenguajes de Shader 751
26.5. Desarrollo de shaders en Ogre 751
26.5.1. Primer Shader 752
26.5.2. Comprobando la interpolación del color 756
26.5.3. Usando una textura 757
26.5.4. Jugando con la textura 759
26.5.5. Jugando con los vértices 762
26.5.6. Iluminación mediante shaders 764
26.6. Optimización de interiores 765
26.6.1. Introducción 765
26.6.2. Técnicas y Algoritmos 766
26.6.3. Manejo de escenas en OGRE 778
26.7. Optimización de Exteriores 780
26.7.1. Introducción 780
26.7.2. Estructuras de datos 780
26.7.3. Determinación de la resolución 782

26.7.4. Técnicas y Algoritmos 784
26.7.5. Exteriores y LOD en OGRE 789
26.7.6. Conclusiones 798

27. Plataformas Móviles **799**
27.1. Método de trabajo con un motor de videojuegos 799
27.1.1. Generación de contenido externo al motor 799
27.1.2. Generación de contenido interno al motor 800
27.2. Creación de escenas 800
27.3. Creación de *prefabs* 803
27.4. Programación de scripts 805
27.4.1. Algunos scripts básicos 805
27.4.2. *Triggers* 806
27.4.3. Invocación de métodos retardada 806
27.4.4. Comunicación entre diferentes scripts 807
27.4.5. Control del flujo general de la partida 808
27.4.6. Programación de enemigos 810
27.4.7. Programación del control del jugador 812
27.4.8. Programación del *interface* 814
27.5. Optimización 816
27.5.1. *Light mapping* 816
27.5.2. *Occlusion culling* 816
27.6. Resultado final 818

IV Desarrollo de Componentes **821**

28. Inteligencia Artificial **823**
28.1. Introducción a la IA para videojuegos 824
28.1.1. Aplicando el Test de Turing 824
28.1.2. Ilusión de inteligencia 825
28.1.3. ¿NPCs o Agentes? 827
28.1.4. Diseño de agentes basado en estados 829
28.1.5. Los lenguajes de *script* 831
28.1.6. Caso de estudio. Un Tetris *inteligente* 832
28.2. Técnicas Fundamentales 834
28.2.1. Lógica Difusa 835
28.2.2. Algoritmos genéticos 846
28.2.3. Redes neuronales 855

28.3. Algoritmos de búsqueda 864
28.3.1. Problemas y soluciones. 864
28.3.2. Organizar la búsqueda de soluciones 867
28.3.3. Búsqueda con información 871
28.4. Planificación de caminos 876
28.4.1. Puntos visibles 876
28.4.2. Problema de búsqueda en grafos 877
28.4.3. Obtención del grafo de puntos visibles 879
28.4.4. Aumentando la resolución 880
28.5. Diseño basado en agentes 883
28.5.1. ¿Qué es un agente? 883
28.5.2. ¿Cómo se construye un agente? 885
28.5.3. Los comportamientos como guía de diseño 888
28.5.4. Implementación de agentes basados en autómatas 897
28.5.5. Usando los objetivos como guía de diseño 909
28.5.6. Reflexiones sobre el futuro 912
28.6. Caso de estudio. Juego deportivo 913
28.6.1. Introducción a *Pygame* 914
28.6.2. Arquitectura propuesta 917
28.6.3. Integración de comportamientos básicos 924
28.6.4. Diseño de la Inteligencia Artificial 928
28.6.5. Consideraciones finales 929
28.7. Sistemas expertos basados en reglas 929
28.7.1. Introducción 929
28.7.2. Caso de estudio: Stratego 931

29. Networking **937**
29.1. Conceptos básicos de redes 937
29.2. Consideraciones de diseño 939
29.3. Eficiencia y limitaciones de la red 942
29.3.1. Peer to peer 942
29.3.2. Cliente-servidor 943
29.3.3. Pool de servidores 943
29.4. Restricciones especificas de los juegos en red 944
29.4.1. Capacidad de cómputo 944
29.4.2. Ancho de banda 944
29.4.3. Latencia 944
29.5. Distribución de información 945
29.6. Modelo de información 946

29.7. Uso de recursos de red 946
29.8. Consistencia e inmediatez 947
29.9. Predicciones y extrapolación 948
29.9.1. Predicción del jugador 948
29.9.2. Predicción del oponente 949
29.10Sockets TCP/IP 951
29.10.1Creación de *Sockets* 951
29.10.2Comunicaciones confiables 954
29.10.3Comunicaciones no confiables 957
29.11*Sockets* TCP/IP avanzados 959
29.11.1Operaciones bloqueantes 961
29.11.2Gestión de buffers 961
29.11.3Serialización de datos 962
29.11.4Multicast y Broadcast 963
29.11.5Opciones de los *sockets* 964
29.12Middlewares de comunicaciones 965
29.12.1ZeroC Ice 966
29.12.2Especificación de interfaces 966
29.12.3Terminología 966
29.12.4«Hola mundo» distribuido 968
29.12.5*twoway*, *oneway* y *datagram* 973
29.12.6Invocación asíncrona 974
29.12.7Propagación de eventos 976

30. Sonido y Multimedia **981**
30.1. Edición de Audio 982
30.1.1. Introducción 982
30.1.2. Herramientas para la creación de audio 985
30.1.3. El proceso creativo 987
30.1.4. Técnicas de creación 990
30.1.5. Programación de audio 992
30.2. Video digital en los videojuegos 994
30.3. Gráficos 3D y video digital 994
30.4. Estándares en video digital 995
30.5. *Plugins* de vídeo para Ogre 996
30.5.1. Instalación de TheoraVideoPlugin 997
30.5.2. Incluyendo vídeo en texturas 997
30.6. Reproducción de vídeo con GStreamer 998
30.6.1. Instalación del *framework* de desarrollo y librerías necesarias 998

30.6.2. Introducción a GStreamer 999
30.6.3. Algunos conceptos básicos 1000
30.6.4. GStreamer en Ogre 1001
30.7. Comentarios finales sobre vídeo digital 1005

31. Interfaces de Usuario Avanzadas 1007
31.1. Introducción a la Visión por Computador 1007
31.2. Introducción a OpenCV 1008
31.2.1. Instalación de OpenCV 1008
31.3. Conceptos previos 1009
31.3.1. Nomenclatura 1009
31.3.2. Interfaces ligeras con HighGUI 1010
31.3.3. Estructura de la imagen 1012
31.3.4. Almacenes de memoria y secuencias 1013
31.4. Primera toma de contacto con OpenCV: mostrando vídeo 1014
31.5. Introducción a los filtros 1015
31.6. Detección de objetos mediante reconocimiento 1018
31.6.1. Trabajando con clasificadores 1018
31.6.2. Interacción con el usuario 1020
31.7. Interacción mediante detección del color de los objetos 1023
31.8. Identificación del movimiento 1025
31.9. Comentarios finales sobre Visión por Computador 1029
31.10Caso de estudio: Wiimote 1029
31.11Descripción del mando de Wii 1029
31.12Librerías para la manipulación del Wiimote 1031
31.13Caso de estudio: Kinect 1038
31.14Comunidad OpenKinect 1039
31.15OpenNI 1040
31.15.1.Instalación 1042
31.15.2.Conceptos iniciales 1042
31.15.3.Manipulando Kinect con OpenNI 1043
31.16Realidad Aumentada 1047
31.16.1.Un poco de historia 1048
31.16.2.Características Generales 1050
31.16.3.Alternativas tecnológicas 1051
31.17ARToolKit 1052
31.17.1.Instalación y configuración 1052
31.18El esperado “Hola Mundo!” 1053
31.18.1.Inicialización 1056

31.18.2.Bucle Principal 1056
31.18.3.Finalización y función Main 1057
31.19Las Entrañas de ARToolKit 1057
31.19.1.Principios Básicos 1058
31.19.2.Calibración de la Cámara 1060
31.19.3.Detección de Marcas 1062
31.20Histórico de Percepciones 1064
31.21Utilización de varios patrones 1067
31.22Relación entre Coordenadas 1070
31.23Integración con OpenCV y Ogre 1073
31.24Consideraciones Finales 1078

Anexos **1079**

A. Introducción a OgreFramework **1081**
A.1. Introducción 1081
A.2. Basic OgreFramework 1082
A.2.1. Arquitectura 1082
A.3. Advanced OgreFramework 1084
A.3.1. Sistema de Estados 1084
A.3.2. Arquitectura 1085
A.4. SdkTrays 1088
A.4.1. Introducción 1089
A.4.2. Requisitos 1089
A.4.3. SdkTrayManager 1089
A.4.4. Widgets 1089
A.4.5. SdkTrayListener 1089
A.4.6. Skins 1090
A.4.7. OgreBites 1090
A.5. btOgre 1090
A.6. Referencias 1092

B. Vault Defense al detalle **1093**
B.1. Configuración en Visual 2010 Express 1093
B.1.1. Compilando CEGUI en Windows 1094
B.1.2. Configurando OGRE en Windows 1095
B.2. GUI en Vault Defense 1100
B.2.1. Botones propios 1100
B.2.2. Pantalla de Carga 1103

B.2.3. Manchas en la cámara 1104
B.3. Luces en Vault Defense: efecto fuego 1106
B.4. Enemigos en Vault Defense 1107

Listado de acrónimos

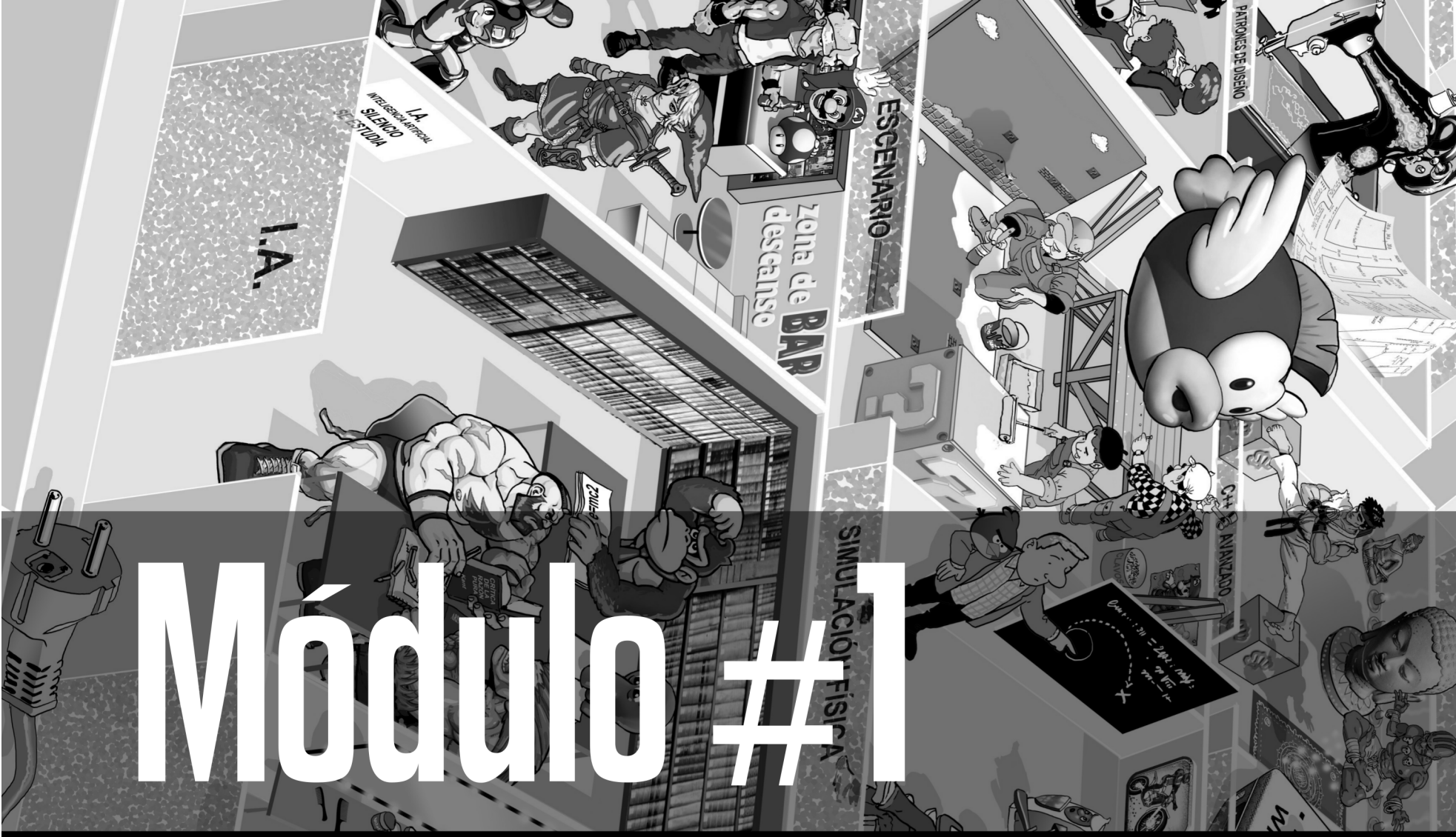
ACE	Adaptive Communications Environment
AMI	Asynchronous Method Invocation
API	Application Program Interface
APT	Advanced Packaging Tool
ARM	Advanced RISC Machine
AVI	Audio Video Interleave
AVL	Adelson-Velskii and Landis
B-REP	Boundary Representation
BBT	Balanced Binary Tree
BDI	Belief, Desire, Intention
BIK	BINK Video
BMP	BitMaP
BRDF	Bidirectional Reflactance Distribution Function
BSD	Berkeley Software Distribution
BSP	Binary Space Partitioning
BTT	Binary Triangle Tree
CAD	Computer Aided Design
CCD	Charge-Coupled Device
CD	Compact Disc
CEGUI	Crazy Eddie's GUI
CMOS	Complementary Metal-Oxide-Semiconductor
CMYK	Cyan Magenta Yellow Key
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CRTP	Curiously Recurring Template Pattern
CSG	Constructive Solid Geometry
CUDA	Compute Unified Device Architecture
CVS	Concurrent Versions System
DBTS	Debian Bug Tracking System
DCU	Diseño Centrado en el Usuario

DD	Debian Developer
DEB	Deep Estimation Buffer
DEHS	Debian External Health Status
DFSG	Debian Free Software Guidelines
DIP	Dependency Inversion Principle
DLL	Dynamic Link Library
DOD	Diamond Of Death
DOM	Document Object Model
DTD	Documento Técnico de Diseño
DVB	Digital Video Broadcasting
DVD	Digital Video Disc
E/S	Entrada/Salida
EASTL	Electronic Arts Standard Template Library
EA	Electronic Arts
ELF	Executable and Linkable Format
FIFO	First In, First Out
FLAC	Free Lossless Audio Codec
FPS	First Person Shooter
FSM	Finite State Machine
GCC	GNU Compiler Collection
GDB	GNU Debugger
GDD	Game Design Document
GIMP	GNU Image Manipulation Program
GLUT	OpenGL Utility Toolkit
GLU	OpenGL Utility
GNOME	GNU Object Model Environment
GNU	GNU is Not Unix
GPL	General Public License
GPS	Global Positioning System
GPU	Graphic Processing Unit
GPV	grafos de puntos visibles
GTK	GIMP ToolKit
GUI	Graphical User Interface
GUP	Game Unified Process
HDTV	High Definition Television
HFSM	Hierarchical Finite State Machine
HOM	Hierarchical Occlusion Maps
HSV	Hue, Saturation, Value
HTML	HyperText Markup Language
I/O	Input/Output
IANA	Internet Assigned Numbers Authority
IA	Inteligencia Artificial
IBM	International Business Machines
IDL	Interface Definition Language

IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IGC	In-Game Cinematics
IP	Internet Protocol
IPC	InterProcess Communication
IPO	InterPOlation curve
IPP	Integraed Performance Primitives
IPX	Internetwork Packet Exchange
ITP	Intent to Package
ISO	International Organization for Standardization
ISP	Interface Segregation Principle
ICE	Internet Communication Engine
JPG	Joint Photographic Experts Group
KISS	Keep it simple, Stupid!
LAN	Local Area Network
LED	Light Emitter Diode
LGPL	Lesser General Public License
LIFO	Last In, First Out
LOD	Level-Of-Detail
LSP	Liskov Substitution Principle
MAC	Media Access Control
MAS	Multi-Agent Systems
MHI	Motion History Image
MMOG	Massively Multiplayer Online Game
MMORPG	Massively Multiplayer Online Role-Playing Game
MP3	MPEG-2 Audio Layer III
MPEG	Moving Picture Experts Group
MTU	Maximum Transfer Unit
MVC	Model View Controller
NPC	Non-Player Character
NRVO	Named Return Value Optimization
OCP	Open/Closed Principle
ODE	Open Dynamics Engine
ODT	OpenDocument Text
OGI	Oregon Graduate Institute
OGRE	Object-Oriented Graphics Rendering Engine
OIS	Object Oriented Input System
ONC	Open Network Computing
P2P	Peer To Peer
PDA	Personal Digital Assistant
PDF	Portable Document Format
PHP	Personal Home Page
PMU	Performance Monitoring Units
PNG	Portable Network Graphics

POO	Programación Orientada a Objetos
POSIX	Portable Operating System Interface X
PPC	PowerPC
PUD	Proceso Unificado de Desarrollo
PVS	Potential Visibility Set
RAII	Resource Acquisition Is Initialization
RFA	Request For Adoption
RFH	Request For Help
RFP	Request For Package
RGBA	Red Green Blue Alpha
RGB	Red Green Blue
RMI	Remote Method Invocation
ROAM	Real-time Optimally Adapting Meshes
ROI	Region Of Interest
RPC	Remote Procedure Call
RPG	Role-Playing Games
RPM	RPM Package Manager
RST	reStructuredText
RTP	Real Time Protocol
RTS	Real-Time Strategy
RTTI	Run Time Type Information
RTT	Render To Texture
RVO	Return Value Optimization
SAX	Simple API for XML
SDK	Software Development Kit
SDL	Simple Directmedia Layer
SGBD	Sistema de Gestión de Base de Datos
SGML	Standard Generalized Markup Language
SGI	Silicon Graphics Incorporated
SIP	Session Initiation Protocol
SLERP	spherical linear interpolation
SOLID	SRP, OCP, LSP, ISP, DIP
SRP	Single responsibility principle
SRU	Sistema de Referencia Universal
STL	Standard Template Library
SUT	Subject Under Test
SVCD	Super VCD
SVG	Scalable Vector Graphics
SLICE	Specification Language for Ice
TCP/IP	Pila de protocolos de Internet
TCP	Transport Control Protocol
TDD	Test Driven Development
TDT	Televisión Digital Terrestre
TGA	Truevision Graphics Adapter

TIFF	Tagged Image File Format
TIF	TIFF
TLB	Translation Lookaside Buffer
TTL	Time To Live
UCLM	Universidad de Castilla-La Mancha
UDP	User Datagram Protocol
UML	Unified Modeling Language
URL	Uniform Resource Locator
USB	Universal Serial Bus
UTF	Unicode Transformation Format
UUID	Universally Unique Identifier
VCD	Video CD
VCS	Version Control System
VGA	Video Graphics Adapter
VRML	Virtual Reality Modeling Language
WAV	WAVeform
WNPP	Work-Needing and Prospective Packages
XDR	eXternal Data Representation
XML	eXtensible Markup Language
YAGNI	You Ain't Gonna Need It
YAML	YAML Ain't Markup Language



Módulo #1

Arquitectura del Motor

Arquitectura del Motor

El objetivo de este módulo, titulado «*Arquitectura del Motor*» dentro del Curso de Experto en Desarrollo de Videojuegos, es introducir los conceptos básicos relativos a las estructuras y principios de diseño y desarrollo comúnmente empleados en la creación de videojuegos. Para ello, uno de los principales objetivos es proporcionar una visión general de la arquitectura general de un motor de juegos. Dentro del contexto de esta arquitectura general se hace especial hincapié en aspectos como los subsistemas de bajo nivel, el bucle de juego, la gestión básica de recursos, como el sonido, y la gestión de la concurrencia. Para llevar a cabo una discusión práctica de todos estos elementos se hace uso del motor de renderizado Ogre3D. Por otra parte, en este primer módulo también se estudian los fundamentos del lenguaje de programación C++ como herramienta esencial para el desarrollo de videojuegos profesionales. Este estudio se complementa con una discusión en profundidad de una gran variedad de patrones de diseño y de la biblioteca STL. Además, también se realiza un recorrido por herramientas que son esenciales en el desarrollo de proyectos software complejos, como por ejemplo los sistemas de control de versiones, o procesos como la compilación o la depuración.

Capítulo 1

Introducción

David Vallejo Fernández

Actualmente, la industria del videojuego goza de una muy buena salud a nivel mundial, rivalizando en presupuesto con las industrias cinematográfica y musical. En este capítulo se discute, desde una perspectiva general, el **desarrollo de videojuegos**, haciendo especial hincapié en su evolución y en los distintos elementos involucrados en este complejo proceso de desarrollo. En la segunda parte del capítulo se introduce el concepto de **arquitectura del motor**, como eje fundamental para el diseño y desarrollo de videojuegos comerciales.

1.1. El desarrollo de videojuegos

1.1.1. La industria del videojuego. Presente y futuro

El primer videojuego

El videojuego *Pong* se considera como unos de los primeros videojuegos de la historia. Desarrollado por Atari en 1975, el juego iba incluido en la consola *Atari Pong*. Se calcula que se vendieron unas 50.000 unidades.

Lejos han quedado los días desde el desarrollo de los primeros videojuegos, caracterizados principalmente por su simplicidad y por el hecho de estar desarrollados completamente sobre hardware. Debido a los distintos avances en el campo de la informática, no sólo a nivel de desarrollo software y capacidad hardware sino también en la aplicación de métodos, técnicas y algoritmos, la industria del videojuego ha evolucionado hasta llegar a cotas inimaginables, tanto a nivel de jugabilidad como de calidad gráfica, tan sólo hace unos años.

La **evolución** de la industria de los videojuegos ha estado ligada a una serie de hitos, determinados particularmente por juegos que han marcado un antes y un después, o por fenómenos sociales que han afectado de manera directa a dicha industria. Juegos como *Doom*, *Quake*, *Final Fantasy*, *Zelda*, *Tekken*, *Gran Turismo*, *Metal Gear*, *The Sims* o *World of Warcraft*, entre otros, han marcado tendencia y han contribuido de manera significativa al desarrollo de videojuegos en distintos géneros.

Por otra parte, y de manera complementaria a la aparición de estas obras de arte, la propia evolución de la informática ha posibilitado la vertiginosa evolución del desarrollo de videojuegos. Algunos **hitos clave** son por ejemplo el uso de la tecnología poligonal en 3D [6] en las consolas de sobremesa, el *boom* de los ordenadores personales como plataforma multi-propósito, la expansión de Internet, los avances en el desarrollo de microprocesadores, el uso de *shaders* programables [76], el desarrollo de motores de juegos o, más recientemente, la eclosión de las redes sociales y el uso masivo de dispositivos móviles.

Por todo ello, los videojuegos se pueden encontrar en ordenadores personales, consolas de juego de sobremesa, consolas portátiles, dispositivos móviles como por ejemplo los *smartphones*, o incluso en las redes sociales como medio de soporte para el entretenimiento de cualquier tipo de usuario. Esta diversidad también está especialmente ligada a distintos tipos o géneros de videojuegos, como se introducirá más adelante en esta misma sección.

La **expansión del videojuego** es tan relevante que actualmente se trata de una industria multimillonaria capaz de rivalizar con las industrias cinematográfica y musical. Un ejemplo representativo es el valor total del mercado del videojuego en Europa, tanto a nivel hardware como software, el cual alcanzó la nada desdeñable cifra de casi 11.000 millones de euros, con países como Reino Unido, Francia o Alemania a la cabeza. En este contexto, España representa el cuarto consumidor a nivel europeo y también ocupa una posición destacada dentro del *ranking* mundial.

A pesar de la vertiginosa evolución de la industria del videojuego, hoy en día existe un gran número de **retos** que el desarrollador de videojuegos ha de afrontar a la hora de producir un videojuego. En realidad, existen retos que perdurarán eternamente y que no están ligados a la propia evolución del hardware que permite la ejecución de los videojuegos. El más evidente de ellos es la necesidad imperiosa de ofrecer una experiencia de entretenimiento al usuario basada en la diversión, ya sea a través de nuevas formas de interacción, como por ejemplo la realidad aumentada o la tecnología de visualización 3D, a través de una mejora evidente en la calidad de los títulos, o mediante innovación en aspectos vinculados a la jugabilidad.

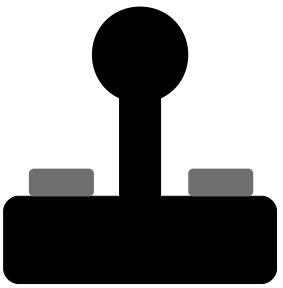


Figura 1.1: El desarrollo y la innovación en hardware también supone un pilar fundamental en la industria del videojuego.

No obstante, actualmente la evolución de los videojuegos está estrechamente ligada a la **evolución del hardware** que permite la ejecución de los mismos. Esta evolución atiende, principalmente, a dos factores: i) la potencia de dicho hardware y ii) las capacidades interactivas del mismo. En el primer caso, una mayor potencia hardware implica que el desarrollador disfrute de mayores posibilidades a la hora de, por ejemplo, mejorar la calidad gráfica de un título o de incrementar la IA (Inteligencia Artificial) de los enemigos. Este factor está vinculado al **multiprocesamiento**. En el segundo caso, una mayor riqueza en términos de interactividad puede contribuir a que el usuario de videojuegos viva una experiencia más inmersiva (por ejemplo, mediante realidad aumentada) o, simplemente, más natural (por ejemplo, mediante la pantalla táctil de un *smartphone*).

Finalmente, resulta especialmente importante destacar la existencia de **motores de juego** (*game engines*), como por ejemplo *Quake*[1] o *Unreal*[2], *middlewares* para el tratamiento de aspectos específicos de un juego, como por ejemplo la biblioteca *Havok*[3] para el tratamiento de la física, o motores de renderizado, como por ejemplo *Ogre 3D* [52]. Este tipo de herramientas, junto con técnicas específicas de desarrollo y optimización, metodologías de desarrollo, o patrones de diseño, entre otros, conforman un aspecto esencial a la hora de desarrollar un videojuego. Al igual que ocurre en otros aspectos relacionados con la Ingeniería del Software, desde un punto de vista general resulta aconsejable el uso de todos estos elementos para agilizar el proceso de desarrollo y reducir errores potenciales. En otras palabras, no es necesario, ni productivo, reinventar la rueda cada vez que se afronta un nuevo proyecto.

1.1.2. Estructura típica de un equipo de desarrollo

Tiempo real

En el ámbito del desarrollo de videojuegos, el concepto de tiempo real es muy importante para dotar de realismo a los juegos, pero no es tan estricto como el concepto de tiempo real manejado en los sistemas críticos.

El desarrollo de videojuegos comerciales es un proceso complejo debido a los distintos requisitos que ha de satisfacer y a la integración de distintas disciplinas que intervienen en dicho proceso. Desde un punto de vista general, un videojuego es una **aplicación gráfica en tiempo real** en la que existe una interacción explícita mediante el usuario y el propio videojuego. En este contexto, el concepto de tiempo real se refiere a la necesidad de generar una determinada tasa de *frames* o imágenes por segundo, típicamente 30 ó 60, para que el usuario tenga una sensación continua de realidad. Por otra parte, la interacción se refiere a la forma de comunicación existente entre el usuario y el videojuego. Normalmente, esta interacción se realiza mediante *joysticks* o mandos, pero también es posible llevarla a cabo con otros dispositivos como por ejemplo teclados, ratones, cascos o incluso mediante el propio cuerpo a través de técnicas de visión por computador o de interacción táctil.

A continuación se describe la estructura típica de un equipo de desarrollo atendiendo a los distintos roles que juegan los componentes de dicho equipo [42]. En muchos casos, y en función del número de componentes del equipo, hay personas especializadas en diversas disciplinas de manera simultánea.

Los **ingenieros** son los responsables de diseñar e implementar el software que permite la ejecución del juego, así como las herramientas que dan soporte a dicha ejecución. Normalmente, los ingenieros se suelen clasificar en dos grandes grupos:

- Los **programadores del núcleo** del juego, es decir, las personas responsables de desarrollar tanto el motor de juego como el juego propiamente dicho.
- Los **programadores de herramientas**, es decir, las personas responsables de desarrollar las herramientas que permiten que el resto del equipo de desarrollo pueda trabajar de manera eficiente.

De manera independiente a los dos grupos mencionados, los ingenieros se pueden especializar en una o en varias disciplinas. Por ejemplo, resulta bastante común encontrar perfiles de ingenieros especializados en programación gráfica o en *scripting* e IA. Sin embargo, tal y como se sugirió anteriormente, el concepto de *ingeniero transversal* es bastante común, particularmente en equipos de desarrollo que tienen un número reducido de componentes o con un presupuesto que no les permite la contratación de personas especializadas en una única disciplina.

[1] http://www.idsoftware.com/games/quake/quake/
[2] http://www.unrealengine.com/
[3] http://www.havok.com

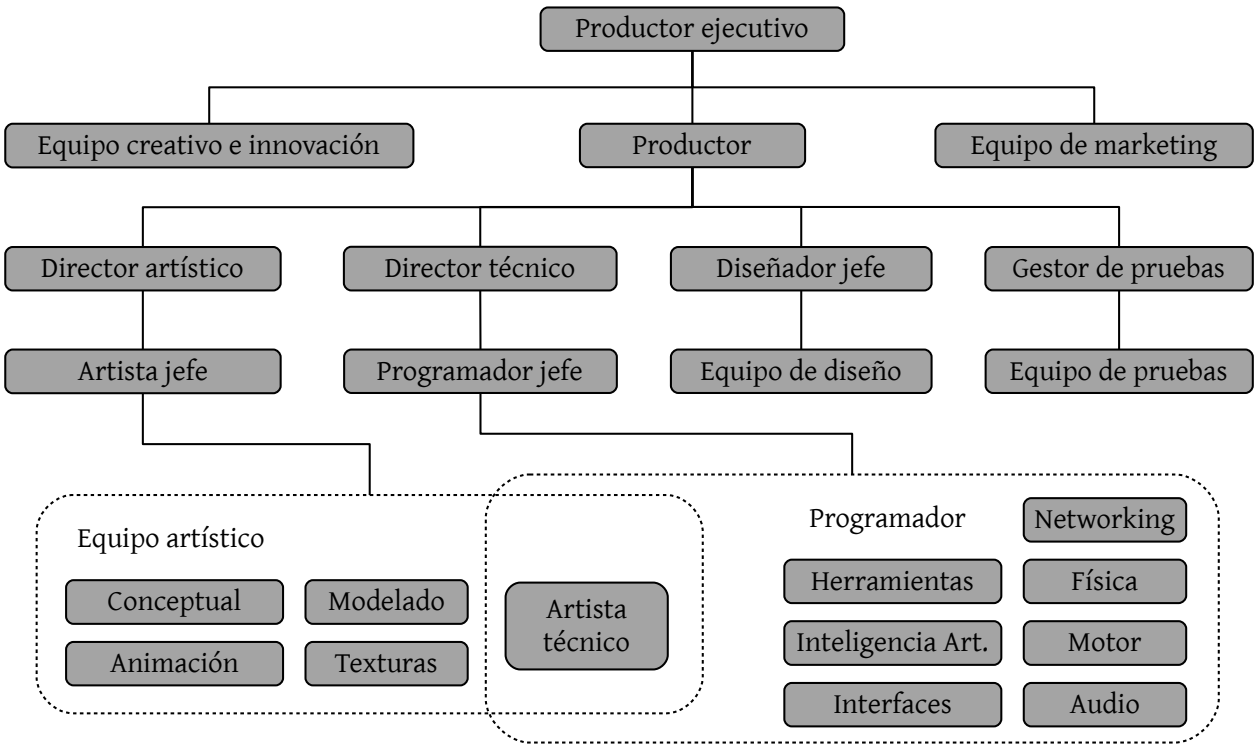


Figura 1.2: Visión conceptual de un equipo de desarrollo de videojuegos, considerando especialmente la parte de programación.

En el mundo del desarrollo de videojuegos, es bastante probable encontrar ingenieros *senior* responsables de supervisar el desarrollo desde un punto de vista técnico, de manera independiente al diseño y generación de código. No obstante, este tipo de roles suelen estar asociados a la supervisión técnica, la gestión del proyecto e incluso a la toma de decisiones vinculadas a la dirección del proyecto. Así mismo, algunas compañías también pueden tener directores técnicos, responsables de la supervisión de uno o varios proyectos, e incluso un director ejecutivo, encargado de ser el director técnico del estudio completo y de mantener, normalmente, un rol ejecutivo en la compañía o empresa.

Los **artistas** son los responsables de la creación de todo el contenido audio-visual del videojuego, como por ejemplo los escenarios, los personajes, las animaciones de dichos personajes, etc. Al igual que ocurre en el caso de los ingenieros, los artistas también se pueden especializar en diversas cuestiones, destacando las siguientes:

General VS Específico

En función del tamaño de una empresa de desarrollo de videojuegos, el nivel de especialización de sus empleados es mayor o menor. Sin embargo, las ofertas de trabajo suelen incluir diversas disciplinas de trabajo para facilitar su integración.

- Artistas de concepto, responsables de crear bocetos que permitan al resto del equipo hacerse una idea inicial del aspecto final del videojuego. Su trabajo resulta especialmente importante en las primeras fases de un proyecto.
- Modeladores, responsables de generar el contenido 3D del videojuego, como por ejemplo los escenarios o los propios personajes que forman parte del mismo.
- Artistas de texturizado, responsables de crear las texturas o imágenes bidimensionales que formarán parte del contenido visual del juego. Las texturas se aplican sobre la geometría de los modelos con el objetivo de dotarlos de mayor realismo.

- Artistas de iluminación, responsables de gestionar las fuentes de luz del videojuego, así como sus principales propiedades, tanto estáticas como dinámicas.
- Animadores, responsables de dotar de movimientos a los personajes y objetos dinámicos del videojuego. Un ejemplo típico de animación podría ser el movimiento de brazos de un determinado carácter.
- Actores de captura de movimiento, responsables de obtener datos de movimiento reales para que los animadores puedan integrarlos a la hora de animar los personajes.
- Diseñadores de sonido, responsables de integrar los efectos de sonido del videojuego.
- Otros actores, responsables de diversas tareas como por ejemplo los encargados de dotar de voz a los personajes.

Al igual que suele ocurrir con los ingenieros, existe el rol de artista *senior* cuyas responsabilidades también incluyen la supervisión de los numerosos aspectos vinculados al componente artístico.

Scripting e IA

El uso de lenguajes de alto nivel es bastante común en el desarrollo de videojuegos y permite diferenciar claramente la lógica de la aplicación y la propia implementación. Una parte significativa de las desarrolladoras utiliza su propio lenguaje de *scripting*, aunque existen lenguajes ampliamente utilizados, como son *Lua* o *Python*.

Los **diseñadores de juego** son los responsables de diseñar el contenido del juego, destacando la evolución del mismo desde el principio hasta el final, la secuencia de capítulos, las reglas del juego, los objetivos principales y secundarios, etc. Evidentemente, todos los aspectos de diseño están estrechamente ligados al propio género del mismo. Por ejemplo, en un juego de conducción es tarea de los diseñadores definir el comportamiento de los coches adversarios ante, por ejemplo, el adelantamiento de un rival.

Los diseñadores suelen trabajar directamente con los ingenieros para afrontar diversos retos, como por ejemplo el comportamiento de los enemigos en una aventura. De hecho, es bastante común que los propios diseñadores programen, junto con los ingenieros, dichos aspectos haciendo uso de lenguajes de *scripting* de alto nivel, como por ejemplo *Lua*[4] o *Python*[5].

Como ocurre con las otras disciplinas previamente comentadas, en algunos estudios los diseñadores de juego también juegan roles de gestión y supervisión técnica.

Finalmente, en el desarrollo de videojuegos también están presentes roles vinculados a la producción, especialmente en estudios de mayor capacidad, asociados a la planificación del proyecto y a la gestión de recursos humanos. En algunas ocasiones, los productores también asumen roles relacionados con el diseño del juego. Así mismo, los responsables de *marketing*, de administración y de soporte juegan un papel relevante. También resulta importante resaltar la figura de publicador como entidad responsable del *marketing* y distribución del videojuego desarrollado por un determinado estudio. Mientras algunos estudios tienen contratos permanentes con un determinado publicador, otros prefieren mantener una relación temporal y asociarse con el publicador que le ofrezca mejores condiciones para gestionar el lanzamiento de un título.

1.1.3. El concepto de juego

Dentro del mundo del entretenimiento electrónico, un **juego** normalmente se suele asociar a la evolución, entendida desde un punto de vista general, de uno o varios personajes principales o entidades que pretenden alcanzar una serie de objetivos en un mundo acotado, los cuales están controlados por el propio usuario. Así, entre estos

[4] http://www.lua.org
[5] http://www.python.org

elementos podemos encontrar desde superhéroes hasta coches de competición pasando por equipos completos de fútbol. El mundo en el que conviven dichos personajes suele estar compuesto, normalmente, por una serie de escenarios virtuales recreados en tres dimensiones y tiene asociado una serie de reglas que determinan la interacción con el mismo.

De este modo, existe una **interacción** explícita entre el jugador o usuario de videojuegos y el propio videojuego, el cual plantea una serie de retos al usuario con el objetivo final de garantizar la diversión y el entretenimiento. Además de ofrecer este componente emocional, los videojuegos también suelen tener un componente cognitivo asociado, obligando a los jugadores a aprender técnicas y a dominar el comportamiento del personaje que manejan para resolver los retos o puzzles que los videojuegos plantean.

Desde una perspectiva más formal, la mayoría de videojuegos suponen un ejemplo representativo de lo que se define como aplicaciones gráficas o **renderizado en tiempo real** [6], las cuales se definen a su vez como la rama más interactiva de la Informática Gráfica. Desde un punto de vista abstracto, una aplicación gráfica en tiempo real se basa en un bucle donde en cada iteración se realizan los siguientes pasos:

- El usuario visualiza una imagen renderizada por la aplicación en la pantalla o dispositivo de visualización.
- El usuario actúa en función de lo que haya visualizado, interactuando directamente con la aplicación, por ejemplo mediante un teclado.
- En función de la acción realizada por el usuario, la aplicación gráfica genera una salida u otra, es decir, existe una retroalimentación que afecta a la propia aplicación.

Caída de frames

Si el núcleo de ejecución de un juego no es capaz de mantener los *fps* a un nivel constante, el juego sufrirá una caída de frames en un momento determinado. Este hecho se denomina comúnmente como *ralentización*.

En el caso de los videojuegos, este ciclo de visualización, actuación y renderizado ha de ejecutarse con una frecuencia lo suficientemente elevada como para que el usuario se sienta inmerso en el videojuego, y no lo perciba simplemente como una sucesión de imágenes estáticas. En este contexto, el **frame rate** se define como el número de imágenes por segundo, comúnmente *fps*, que la aplicación gráfica es capaz de generar. A mayor *frame rate*, mayor sensación de realismo en el videojuego. Actualmente, una tasa de 30 *fps* se considera más que aceptable para la mayoría de juegos. No obstante, algunos juegos ofrecen tasas que doblan dicha medida.

Generalmente, el desarrollador de videojuegos ha de buscar un compromiso entre los *fps* y el grado de realismo del videojuego. Por ejemplo, el uso de modelos con una alta complejidad computacional, es decir, con un mayor número de polígonos, o la integración de comportamientos inteligentes por parte de los enemigos en un juego, o NPC (Non-Player Character), disminuirá los *fps*.

En otras palabras, los juegos son aplicaciones interactivas que están marcadas por el tiempo, es decir, cada uno de los ciclos de ejecución tiene un *deadline* que ha de cumplirse para no perder realismo.

Aunque el **componente gráfico** representa gran parte de la complejidad computacional de los videojuegos, no es el único. En cada ciclo de ejecución, el videojuego ha de tener en cuenta la evolución del mundo en el que se desarrolla el mismo. Dicha evolución dependerá del estado de dicho mundo en un momento determinado y de cómo las distintas entidades dinámicas interactúan con él. Obviamente, recrear el mundo

real con un nivel de exactitud elevado no resulta manejable ni práctico, por lo que normalmente dicho mundo se aproxima y se simplifica, utilizando modelos matemáticos para tratar con su complejidad. En este contexto, destaca por ejemplo la simulación física de los propios elementos que forman parte del mundo.

Figura 1.3: El motor de juego representa el núcleo de un videojuego y determina el comportamiento de los distintos módulos que lo componen.

Por otra parte, un juego también está ligado al comportamiento del personaje principal y del resto de entidades que existen dentro del mundo virtual. En el ámbito académico, estas entidades se suelen definir como **agentes** (*agents*) y se encuadran dentro de la denominada *simulación basada en agentes* [64]. Básicamente, este tipo de aproximaciones tiene como objetivo dotar a los NPC con cierta inteligencia para incrementar el grado de realismo de un juego estableciendo, incluso, mecanismos de cooperación y coordinación entre los mismos. Respecto al personaje principal, un videojuego ha de contemplar las distintas acciones realizadas por el mismo, considerando la posibilidad de decisiones impredecibles a priori y las consecuencias que podrían desencadenar.

En resumen, y desde un punto de vista general, el desarrollo de un juego implica considerar un gran número de factores que, inevitablemente, incrementan la complejidad del mismo y, al mismo tiempo, garantizar una tasa de *fps* adecuada para que la inmersión del usuario no se vea afectada.

1.1.4. Motor de juego

Al igual que ocurre en otras disciplinas en el campo de la informática, el desarrollo de videojuegos se ha beneficiado de la aparición de herramientas que facilitan dicho desarrollo, automatizando determinadas tareas y ocultando la complejidad inherente a muchos procesos de bajo nivel. Si, por ejemplo, los SGBD han facilitado enormemente la gestión de persistencia de innumerables aplicaciones informáticas, los motores de juegos hacen la vida más sencilla a los desarrolladores de videojuegos.

Según [42], el término *motor de juego* surgió a mediados de los años 90 con la aparición del famosísimo juego de acción en primera persona *Doom*, desarrollado por la compañía *id Software* bajo la dirección de *John Carmack*[6]. Esta afirmación se sustenta sobre el hecho de que *Doom* fue diseñado con una **arquitectura orientada a la reutilización** mediante una separación adecuada en distintos módulos de los componentes fundamentales, como por ejemplo el sistema de renderizado gráfico, el sistema de detección de colisiones o el sistema de audio, y los elementos más *artísticos*, como por ejemplo los escenarios virtuales o las reglas que gobernaban al propio juego.

Figura 1.4: John Carmack, uno de los desarrolladores de juegos más importantes, en el *Game Developer Conference* del año 2010.

Este planteamiento facilitaba enormemente la reutilización de software y el concepto de motor de juego se hizo más popular a medida que otros desarrolladores comenzaron a utilizar diversos módulos o juegos previamente licenciados para generar los suyos propios. En otras palabras, era posible diseñar un juego del mismo tipo sin apenas modificar el núcleo o *motor* del juego, sino que el esfuerzo se podía dirigir directamente a la parte artística y a las reglas del mismo.

Este enfoque ha ido evolucionando y se ha expandido, desde la generación de ***mods*** por desarrolladores independientes o *amateurs* hasta la creación de una gran variedad de herramientas, bibliotecas e incluso lenguajes que facilitan el desarrollo de videojuegos. A día de hoy, una gran parte de compañías de desarrollo de videojuego utilizan motores o herramientas pertenecientes a terceras partes, debido a que les resulta más rentable económicamente y obtienen, generalmente, resultados espectaculares. Por otra parte, esta evolución también ha permitido que los desarrolladores de un juego se planteen licenciar parte de su propio motor de juego, decisión que también forma parte de su política de trabajo.

[6] http://en.wikipedia.org/wiki/John_D._Carmack

Obviamente, la separación entre motor de juego y juego nunca es total y, por una circunstancia u otra, siempre existen dependencias directas que no permiten la reusabilidad completa del motor para crear otro juego. La dependencia más evidente es el genero al que está vinculado el motor de juego. Por ejemplo, un motor de juegos diseñado para construir juegos de acción en primera persona, conocidos tradicionalmente como *shooters* o *shoot'em all*, será difícilmente reutilizable para desarrollar un juego de conducción.

Una forma posible para diferenciar un motor de juego y el software que representa a un juego está asociada al concepto de **arquitectura dirigida por datos** (*data-driven architecture*). Básicamente, cuando un juego contiene parte de su lógica o funcionamiento en el propio código (*hard-coded logic*), entonces no resulta práctico reutilizarla para otro juego, ya que implicaría modificar el código fuente sustancialmente. Sin embargo, si dicha lógica o comportamiento no está definido a nivel de código, sino por ejemplo mediante una serie de reglas definidas a través de un lenguaje de *script*, entonces la reutilización sí es posible y, por lo tanto, beneficiosa, ya que optimiza el tiempo de desarrollo.

Como conclusión final, resulta relevante destacar la evolución relativa a la generalidad de los motores de juego, ya que poco a poco están haciendo posible su utilización para diversos tipos de juegos. Sin embargo, el compromiso entre generalidad y optimalidad aún está presente. En otras palabras, a la hora de desarrollar un juego utilizando un determinado motor es bastante común personalizar dicho motor para adaptarlo a las necesidades concretas del juego a desarrollar.

Game engine *tuning*

Los motores de juegos se suelen adaptar para cubrir las necesidades específicas de un título y para obtener un mejor rendimiento.

1.1.5. Géneros de juegos

Los motores de juegos suelen estar, generalmente, ligados a un tipo o género particular de juegos. Por ejemplo, un motor de juegos diseñado con la idea de desarrollar juegos de conducción diferirá en gran parte con respecto a un motor orientado a juegos de acción en tercera persona. No obstante, y tal y como se discutirá en la sección 1.2, existen ciertos módulos, sobre todo relativos al procesamiento de más bajo nivel, que son transversales a cualquier tipo de juego, es decir, que se pueden reutilizar en gran medida de manera independiente al género al que pertenezca el motor. Un ejemplo representativo podría ser el módulo de tratamiento de eventos de usuario, es decir, el módulo responsable de recoger y gestionar la interacción del usuario a través de dispositivos como el teclado, el ratón, el joystick o la pantalla táctil. Otros ejemplo podría ser el módulo de tratamiento del audio o el módulo de renderizado de texto.

A continuación, se realizará una descripción de los distintos géneros de juegos más populares atendiendo a las características que diferencian unos de otros en base al motor que les da soporte. Esta descripción resulta útil para que el desarrollador identifique los aspectos críticos de cada juego y utilice las técnicas de desarrollo adecuadas para obtener un buen resultado.

Probablemente, el género de juegos más popular ha sido y es el de los los denominados FPS, abreviado tradicionalmente como *shooters*, representado por juegos como *Quake*, *Half-Life*, *Call of Duty* o *Gears of War*, entre muchos otros. En este género, el usuario normalmente controla a un personaje con una vista en primera persona a lo largo de escenarios que tradicionalmente han sido interiores, como los típicos pasillos, pero que han ido evolucionando a escenarios exteriores de gran complejidad.

Mercado de *shooters*

Los FPS (First Person Shooter) gozan actualmente de un buen momento y, como consecuencia de ello, el número de títulos disponibles es muy elevado, ofreciendo una gran variedad al usuario final.

Los FPS representan juegos con un desarrollo complejo, ya que uno de los retos principales que han de afrontar es la inmersión del usuario en un mundo hiperrealista que ofrezca un alto nivel de detalle, al mismo tiempo que se garantice una alta reacción de respuesta a las acciones del usuario. Este género de juegos se centra en la aplicación de las siguientes tecnologías [42]:

Figura 1.5: Captura de pantalla del juego *Tremulous*®, licenciado bajo GPL y desarrollado sobre el motor de *Quake III*.

- Renderizado eficiente de grandes escenarios virtuales 3D.
- Mecanismo de respuesta eficiente para controlar y apuntar con el personaje.
- Detalle de animación elevado en relación a las armas y los brazos del personaje virtual.
- Uso de una gran variedad de arsenal.
- Sensación de que el personaje *flota* sobre el escenario, debido al movimiento del mismo y al modelo de colisiones.
- NPC con un nivel de IA considerable y dotados de buenas animaciones.
- Inclusión de opciones multijugador a baja escala, típicamente entre 32 y 64 jugadores.

Normalmente, la tecnología de renderizado de los FPS está especialmente optimizada atendiendo, entre otros factores, al tipo de escenario en el que se desarrolla el juego. Por ejemplo, es muy común utilizar estructuras de datos auxiliares para disponer de más información del entorno y, consecuentemente, optimizar el cálculo de diversas tareas. Un ejemplo muy representativo en los escenarios interiores son los árboles BSP (Binary Space Partitioning) (árboles de partición binaria del espacio) [6], que se utilizan para realizar una división del espacio físico en dos partes, de manera recursiva, para optimizar, por ejemplo, aspectos como el cálculo de la posición de un jugador. Otro ejemplo representativo en el caso de los escenarios exteriores es el denominado *occlusion culling* [6], que se utiliza para optimizar el proceso de renderizado descartando aquellos objetos 3D que no se ven desde el punto de vista de la cámara, reduciendo así la carga computacional de dicho proceso.

En el **ámbito comercial**, la familia de motores *Quake*, creados por *Id Software*, se ha utilizado para desarrollar un gran número de juegos, como la saga *Medal of Honor*, e incluso motores de juegos. Hoy es posible descargar el código fuente de *Quake*, *Quake II* y *Quake III* [7] y estudiar su arquitectura para hacerse una idea bastante aproximada de cómo se construyen los motores de juegos actuales.

Otra familia de motores ampliamente conocida es la de *Unreal*, juego desarrollado en 1998 por *Epic Games*. Actualmente, la tecnología *Unreal Engine* se utiliza en multitud de juegos, algunos de ellos tan famosos como *Gears of War*.

Más recientemente, la compañía *Crytek* ha permitido la descarga del CryENGINE 3 SDK (Software Development Kit)[8] para propósitos no comerciales, sino principalmente académicos y con el objetivo de crear una comunidad de desarrollo. Este kit de desarrollo para aplicaciones gráficas en tiempo real es exactamente el mismo que el utilizado por la propia compañía para desarrollar juegos comerciales, como por ejemplo *Crysis 2*.

Otro de los géneros más relevantes son los denominados ***juegos en tercera persona***, donde el usuario tiene el control de un personaje cuyas acciones se pueden apreciar por completo desde el punto de vista de la cámara virtual. Aunque existe un gran parecido entre este género y el de los FPS, los juegos en tercera persona hacen especial hincapié en la animación del personaje, destacando sus movimientos y habilidades, además de prestar mucha atención al detalle gráfico de la totalidad de su cuerpo. Ejemplos representativos de este género son *Resident Evil*, *Metal Gear*, *Gears of War* o *Uncharted*, entre otros.

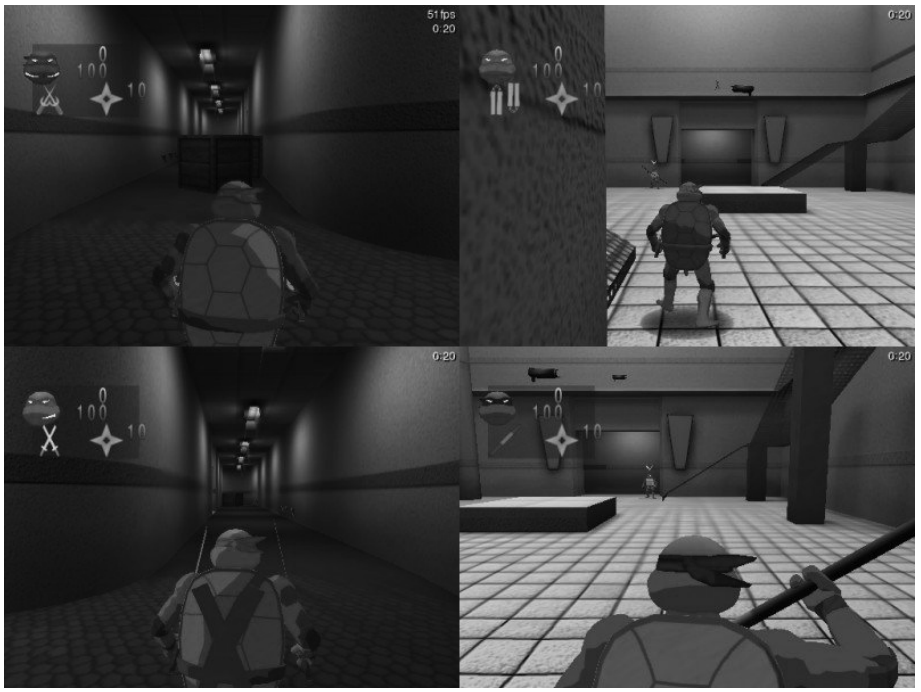


Figura 1.6: Captura de pantalla del juego *Turtlearena*®, licenciado bajo GPL y desarrollado sobre el motor de *Quake III*.

Super Mario Bros

El popular juego de Mario, diseñado en 1985 por Shigeru Miyamoto, ha vendido aproximadamente 40 millones de juegos a nivel mundial. Según el libro de los *Record Guinness*, es una de los juegos más vendidos junto a Tetris y a la saga de Pokemon.

[7] http://www.idsoftware.com/business/techdownloads
[8] `http://mycryengine.com/`

Dentro de este género resulta importante destacar los juegos de plataformas, en los que el personaje principal ha de ir avanzado de un lugar a otro del escenario hasta alcanzar un objetivo. Ejemplos representativos son las sagas de *Super Mario*, *Sonic* o *Donkey Kong*. En el caso particular de los juegos de plataformas, el avatar del personaje tiene normalmente un efecto de *dibujo animado*, es decir, no suele necesitar un renderizado altamente realista y, por lo tanto, complejo. En cualquier caso, la parte dedicada a la animación del personaje ha de estar especialmente cuidada para incrementar la sensación de realismo a la hora de controlarlo.

En los juegos en tercera persona, los desarrolladores han de prestar especial atención a la aplicación de las siguientes tecnologías [42]:

Gráficos 3D

Virtua Fighter, lanzado en 1993 por Sega y desarrollado por Yu Suzuki, se considera como el primer juego de lucha arcade en soportar gráficos tridimensionales.

- Uso de plataformas móviles, equipos de escalado, cuerdas y otros modos de movimiento avanzados.
- Inclusión de puzzles en el desarrollo del juego.
- Uso de cámaras de seguimiento en tercera persona centradas en el personaje y que posibiliten que el propio usuario las maneje a su antojo para facilitar el control del personaje virtual.
- Uso de un complejo sistema de colisiones asociado a la cámara para garantizar que la visión no se vea dificultada por la geometría del entorno o los distintos objetos dinámicos que se mueven por el mismo.

Otro género importante está representado por los **juegos de lucha**, en los que, normalmente, dos jugadores compiten para ganar un determinado número de combates minando la vida o *stamina* del jugador contrario. Ejemplos representativos de juegos de lucha son *Virtua Fighter*, *Street Fighter*, *Tekken*, o *Soul Calibur*, entre otros. Actualmente, los juegos de lucha se desarrollan normalmente en escenarios tridimensionales donde los luchadores tienen una gran libertad de movimiento. Sin embargo, últimamente se han desarrollado diversos juegos en los que tanto el escenario como los personajes son en 3D, pero donde el movimiento de los mismos está limitado a dos dimensiones, enfoque comúnmente conocido como juegos de lucha de *scroll lateral*.

Debido a que en los juegos de lucha la acción se centra generalmente en dos personajes, éstos han de tener una gran calidad gráfica y han de contar con una gran variedad de movimientos y animaciones para dotar al juego del mayor realismo posible. Así mismo, el escenario de lucha suele estar bastante acotado y, por lo tanto, es posible simplificar su tratamiento y, en general, no es necesario utilizar técnicas de optimización como las comentadas en el género de los FPS. Por otra parte, el tratamiento de sonido no resulta tan complejo como lo puede ser en otros géneros de acción.

Los juegos del género de la lucha han de prestar atención a la detección y gestión de colisiones entre los propios luchadores, o entre las armas que utilicen, para dar una sensación de mayor realismo. Además, el módulo responsable del tratamiento de la entrada al usuario ha de ser lo suficientemente sofisticado para gestionar de manera adecuada las distintas combinaciones de botones necesarias para realizar complejos movimientos. Por ejemplo, juegos como *Street Fighter IV* incorporan un sistema de *timing* entre los distintos movimientos de un *combo*. El objetivo perseguido consiste en que dominar completamente a un personaje no sea una tarea sencilla y requiera que el usuario de videojuegos dedique tiempo al entrenaiento del mismo.

Simuladores F1

Los simuladores de juegos de conducción no sólo se utilizan para el entretenimiento doméstico sino también para que, por ejemplo, los pilotos de Fórmula-1 conozcan todos los entresijos de los circuitos y puedan conocerlos al detalle antes de embarcarse en los entrenamientos reales.

Los juegos de lucha, en general, han estado ligados a la evolución de técnicas complejas de síntesis de imagen aplicadas sobre los propios personajes con el objetivo de mejorar al máximo su calidad y, de este modo, incrementar su realismo. Un ejemplo representativo es el uso de *shaders* [76] sobre la armadura o la propia piel de los personajes que permitan implementar técnicas como el *bump mapping* [6], planteada para dotar a estos elementos de un aspecto más rugoso.

Otro género representativo en el mundo de los videojuegos es la **conducción**, en el que el usuario controla a un vehículo que normalmente rivaliza con más adversarios virtuales o reales para llegar a la meta en primera posición. En este género se suele distinguir entre *simuladores*, como por ejemplo *Gran Turismo*, y *arcade*, como por ejemplo *Ridge Racer* o *Wipe Out*.

Figura 1.7: Captura de pantalla del juego de conducción *Tux Racing*, licenciado bajo GPL por Jasmin Patry.

Mientras los simuladores tienen como objetivo principal representar con fidelidad el comportamiento del vehículo y su interacción con el escenario, los juegos arcade se centran más en la jugabilidad para que cualquier tipo de usuario no tenga problemas de conducción.

Los juegos de conducción se caracterizan por la necesidad de dedicar un esfuerzo considerable en alcanzar una calidad gráfica elevada en aquellos elementos cercanos a la cámara, especialmente el propio vehículo. Además, este tipo de juegos, aunque suelen ser muy lineales, mantienen una velocidad de desplazamiento muy elevada, directamente ligada a la del propio vehículo.

Al igual que ocurre en el resto de géneros previamente comentados, existen diversas técnicas que pueden contribuir a mejorar la eficiencia de este tipo de juegos. Por ejemplo, suele ser bastante común utilizar estructuras de datos auxiliares para dividir el escenario en distintos tramos, con el objetivo de optimizar el proceso de renderizado o incluso facilitar el cálculo de rutas óptimas utilizando técnicas de IA [79]. También se suelen usar imágenes para renderizar elementos lejanos, como por ejemplo árboles, vallas publicitarias u otro tipo de elementos.

Del mismo modo, y al igual que ocurre con los juegos en tercera persona, la cámara tiene un papel relevante en el seguimiento del juego. En este contexto, el usuario normalmente tiene la posibilidad de elegir el tipo de cámara más adecuado, como por ejemplo una cámara en primera persona, una en la que se visualicen los controles del propio vehículo o una en tercera persona.

Figura 1.8: Captura de pantalla del juego de estrategia en tiempo real *0 A.D.*, licenciado bajo GPL por *Wildfiregames*.

Otro género tradicional son los juegos de **estrategia**, normalmente clasificados en tiempo real o RTS (Real-Time Strategy)) y por turnos (*turn-based strategy*). Ejemplos representativos de este género son *Warcraft*, *Command & Conquer*, *Comandos*, *Age of Empires* o *Starcraft*, entre otros. Este tipo de juegos se caracterizan por mantener una cámara con una perspectiva isométrica, normalmente fija, de manera que el jugador tiene una visión más o menos completa del escenario, ya sea 2D o 3D. Así mismo, es bastante común encontrar un gran número de unidades virtuales desplegadas en el mapa, siendo responsabilidad del jugador su control, desplazamiento y acción.

Teniendo en cuenta las características generales de este género, es posible plantear diversas optimizaciones. Por ejemplo, una de las aproximaciones más comunes en este tipo de juegos consiste en dividir el escenario en una rejilla o *grid*, con el objetivo de facilitar no sólo el emplazamiento de unidades o edificios, sino también la planificación de movimiento de un lugar del mapa a otro. Por otra parte, las unidades se suelen renderizar con una resolución baja, es decir, con un bajo número de polígonos, con el objetivo de posibilitar el despliegue de un gran número de unidades de manera simultánea.

Finalmente, en los últimos años ha aparecido un género de juegos cuya principal característica es la posibilidad de jugar con un gran número de jugadores reales al mismo tiempo, del orden de cientos o incluso miles de jugadores. Los juegos que se encuadran bajo este género se denominan comúnmente MMOG (Massively Multiplayer Online Game). El ejemplo más representativo de este género es el juego *World of Warcarft*. Debido a la necesidad de soportar un gran número de jugadores en línea, los desarrolladores de este tipo de juegos han de realizar un gran esfuerzo en la parte relativa al *networking*, ya que han de proporcionar un servicio de calidad sobre el que construir su modelo de negocio, el cual suele estar basado en suscripciones mensuales o anuales por parte de los usuarios.

Al igual que ocurre en los juegos de estrategia, los MMOG suelen utilizar personajes virtuales en baja resolución para permitir la aparición de un gran número de ellos en pantalla de manera simultánea.

Además de los distintos géneros mencionados en esta sección, existen algunos más como por ejemplo los juegos deportivos, los juegos de rol o RPG (Role-Playing Games) o los juegos de puzzles.

Antes de pasar a la siguiente sección en la que se discutirá la arquitectura general de un motor de juego, resulta interesante destacar la existencia de algunas **herramientas libres** que se pueden utilizar para la construcción de un motor de juegos. Una de las más populares, y que se utilizará en el presente curso, es OGRE 3D[9]. Básicamente, OGRE es un motor de renderizado 3D bien estructurado y con una curva de aprendizaje adecuada. Aunque OGRE no se puede definir como un motor de juegos completo, sí que proporciona un gran número de módulos que permiten integrar funcionalidades no triviales, como iluminación avanzada o sistemas de animación de caracteres.

1.2. Arquitectura del motor. Visión general

En esta sección se plantea una visión general de la arquitectura de un motor de juegos [42], de manera independiente al género de los mismos, prestando especial importancia a los módulos más relevantes desde el punto de vista del desarrollo de videojuegos.

Como ocurre con la gran mayoría de sistemas software que tienen una complejidad elevada, los motores de juegos se basan en una **arquitectura estructurada en capas**. De este modo, las capas de nivel superior dependen de las capas de nivel inferior, pero no de manera inversa. Este planteamiento permite ir añadiendo capas de manera progresiva y, lo que es más importante, permite modificar determinados aspectos de una capa en concreto sin que el resto de capas inferiores se vean afectadas por dicho cambio.

A continuación, se describen los principales módulos que forman parte de la arquitectura que se expone en la figura 1.9.

1.2.1. Hardware, *drivers* y sistema operativo

La arquitectura *Cell*

En arquitecturas más novedosas, como por ejemplo la arquitectura *Cell* usada en *Playstation 3* y desarrollada por *Sony*, *Toshiba* e IBM, las optimizaciones aplicadas suelen ser más dependientes de la plataforma final.

La capa relativa al **hardware** está vinculada a la plataforma en la que se ejecutará el motor de juego. Por ejemplo, un tipo de plataforma específica podría ser una consola de juegos de sobremesa. Muchos de los principios de diseño y desarrollo son comunes a cualquier videojuego, de manera independiente a la plataforma de despliegue final. Sin embargo, en la práctica los desarrolladores de videojuegos siempre llevan a cabo optimizaciones en el motor de juegos para mejorar la eficiencia del mismo, considerando aquellas cuestiones que son específicas de una determinada plataforma.

La capa de ***drivers*** soporta aquellos componentes software de bajo nivel que permiten la correcta gestión de determinados dispositivos, como por ejemplo las tarjetas de aceleración gráfica o las tarjetas de sonido.

La capa del **sistema operativo** representa la capa de comunicación entre los procesos que se ejecutan en el mismo y los recursos hardware asociados a la plataforma en cuestión. Tradicionalmente, en el mundo de los videojuegos los sistemas operativos se compilan con el propio juego para producir un ejecutable. Sin embargo, las

[9] http://www.ogre3d.org/

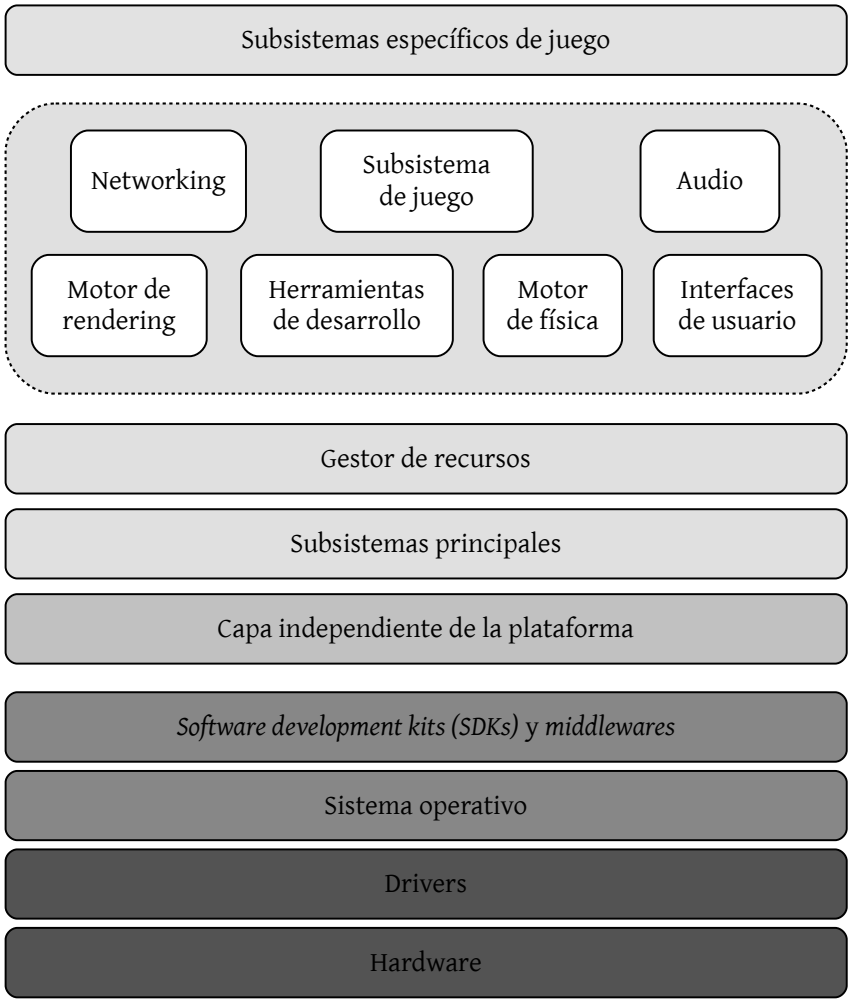


Figura 1.9: Visión conceptual de la arquitectura general de un motor de juegos. Esquema adaptado de la arquitectura propuesta en [42].

consolas de última generación, como por ejemplo *Sony Playstation 3*® o *Microsoft XBox 360*®, incluyen un sistema operativo capaz de controlar ciertos recursos e incluso interrumpir a un juego en ejecución, reduciendo la separación entre consolas de sobremesa y ordenadores personales.

1.2.2. SDKs y *middlewares*

Al igual que ocurre en otros proyectos software, el desarrollo de un motor de juegos se suele apoyar en bibliotecas existentes y SDK para proporcionar una determinada funcionalidad. No obstante, y aunque generalmente este software está bastante optimizado, algunos desarrolladores prefieren personalizarlo para adaptarlo a sus necesidades particulares, especialmente en consolas de sobremesa y portátiles.

APIs gráficas

OpenGL y Direct3D son los dos ejemplos más representativos de API (Application Program Interface)s gráficas que se utilizan en el ámbito comercial. La principal diferencia entre ambas es la estandarización, factor que tiene sus ventajas y desventajas.

Un ejemplo representativo de biblioteca para el manejo de **estructuras de datos** es STL (Standard Template Library) [10]. STL es una biblioteca de plantillas estándar para C++, el cual representa a su vez el lenguaje más extendido actualmente para el desarrollo de videojuegos, debido principalmente a su portabilidad y eficiencia.

En el ámbito de los **gráficos 3D**, existe un gran número de bibliotecas de desarrollo que solventan determinados aspectos que son comunes a la mayoría de los juegos, como el renderizado de modelos tridimensionales. Los ejemplos más representativos en este contexto son las APIs gráficas *OpenGL*[11] y *Direct3D*, mantenidas por el grupo *Khronos* y *Microsoft*, respectivamente. Este tipo de bibliotecas tienen como principal objetivo ocultar los diferentes aspectos de las tarjetas gráficas, presentando una interfaz común. Mientras *OpenGL* es multiplataforma, *Direct3D* está totalmente ligado a sistemas *Windows*.

Otro ejemplo representativo de SDKs vinculados al desarrollo de videojuegos son aquellos que dan soporte a la detección y tratamiento de **colisiones** y a la gestión de la **física** de las distintas entidades que forman parte de un videojuego. Por ejemplo, en el ámbito comercial la compañía *Havok*[12] proporciona diversas herramientas, entre las que destaca *Havok Physics*. Dicha herramienta representa la alternativa comercial más utilizada en el ámbito de la detección de colisiones en tiempo real y en las simulaciones físicas. Según sus autores, *Havok Physics* se ha utilizado en el desarrollo de más de 200 títulos comerciales.

Por otra parte, en el campo del *Open Source*, ODE (Open Dynamics Engine) 3D[13] representa una de las alternativas más populares para simular dinámicas de cuerpo rígido [6].

Recientemente, la rama de la **Inteligencia Artificial** en los videojuegos también se ha visto beneficiada con herramientas que posibilitan la integración directa de bloques de bajo nivel para tratar con problemas clásicos como la búsqueda óptima de caminos entre dos puntos o la acción de evitar obstáculos.

1.2.3. Capa independiente de la plataforma

Abstracción funcional

Aunque en teoría las herramientas multiplataforma deberían abstraer de los aspectos subyacentes a las mismas, como por ejemplo el sistema operativo, en la práctica suele ser necesario realizar algunos ajustos en función de la plataforma existente en capas de nivel inferior.

Gran parte de los juegos se desarrollan teniendo en cuenta su potencial lanzamiento en diversas plataformas. Por ejemplo, un título se puede desarrollar para diversas consolas de sobremesa y para PC al mismo tiempo. En este contexto, es bastante común encontrar una capa software que aisle al resto de capas superiores de cualquier aspecto que sea dependiente de la plataforma. Dicha capa se suele denominar *capa independiente de la plataforma*.

Aunque sería bastante lógico suponer que la capa inmediatamente inferior, es decir, la capa de SDK*s y middleware*, ya posibilita la independencia respecto a las plataformas subyacentes debido al uso de módulos estandarizados, como por ejemplo bibliotecas asociadas a C/C++, la realidad es que existen diferencias incluso en bibliotecas estandarizadas para distintas plataformas.

Algunos ejemplos representativos de módulos incluidos en esta capa son las bibliotecas de manejo de hijos o los *wrappers* o envolturas sobre alguno de los módulos de la capa superior, como el módulo de detección de colisiones o el responsable de la parte gráfica.

[10] `http://www.sgi.com/tech/stl/`
[11] `http://http://www.opengl.org/`
[12] `http://www.havok.com`
[13] `http://www.ode.org`

1.2.4. Subsistemas principales

La capa de subsistemas principales está vinculada a todas aquellas utilidades o bibliotecas de utilidades que dan soporte al motor de juegos. Algunas de ellas son específicas del ámbito de los videojuegos pero otras son comunes a cualquier tipo de proyecto software que tenga una complejidad significativa.

A continuación se enumeran algunos de los subsistemas más relevantes:

- **Biblioteca matemática**, responsable de proporcionar al desarrollador diversas utilidades que faciliten el tratamiento de operaciones relativas a vectores, matrices, cuaterniones u operaciones vinculadas a líneas, rayos, esferas y otras figuras geométricas. Las bibliotecas matemáticas son esenciales en el desarrollo de un motor de juegos, ya que éstos tienen una naturaleza inherentemente matemática.
- **Estructuras de datos y algoritmos**, responsable de proporcionar una implementación más personalizada y optimizada de diversas estructuras de datos, como por ejemplo listas enlazadas o árboles binarios, y algoritmos, como por ejemplo búsqueda u ordenación, que la encontrada en bibliotecas como STL. Este subsistema resulta especialmente importante cuando la memoria de la plataforma o plataformas sobre las que se ejecutará el motor está limitada (como suele ocurrir en consolas de sobremesa).
- **Gestión de memoria**, responsable de garantizar la asignación y liberación de memoria de una manera eficiente.
- **Depuración y *logging***, responsable de proporcionar herramientas para facilitar la depuración y el volcado de *logs* para su posterior análisis.

1.2.5. Gestor de recursos

Esta capa es la responsable de proporcionar una interfaz unificada para acceder a las distintas entidades software que conforman el motor de juegos, como por ejemplo la escena o los propios objetos 3D. En este contexto, existen dos aproximaciones principales respecto a dicho acceso: i) plantear el gestor de recursos mediante un enfoque centralizado y consistente o ii) dejar en manos del programador dicha interacción mediante el uso de archivos en disco.

La figura 1.10 muestra una visión general de un gestor de recursos, representando una interfaz común para la gestión de diversas entidades como por ejemplo el mundo en el que se desarrolla el juego, los objetos 3D, las texturas o los materiales.

En el caso particular de *Ogre 3D* [52], el gestor de recursos está representado por la clase *Ogre::ResourceManager*, tal y como se puede apreciar en la figura 1.11. Dicha clase mantiene diversas especializaciones, las cuales están ligadas a las distintas entidades que a su vez gestionan distintos aspectos en un juego, como por ejemplo las texturas (clase *Ogre::TextureManager*), los modelos 3D (clase *Ogre::MeshManager*) o las fuentes de texto (clase *Ogre::FontManager*). En el caso particular de Ogre 3D, la clase *Ogre::ResourceManager* hereda de dos clases, *ResourceAlloc* y *Ogre::ScriptLoader*, con el objetivo de unificar completamente las diversas gestiones. Por ejemplo, la clase *Ogre::ScriptLoader* posibilita la carga de algunos recursos, como los materiales, mediante *scripts* y, por ello, *Ogre::ResourceManager* hereda de dicha clase.

Ogre 3D

El motor de *rendering* Ogre 3D está escrito en C++ y permite que el desarrollador se abstraiga de un gran número de aspectos relativos al desarrollo de aplicaciones gráficas. Sin embargo, es necesario estudiar su funcionamiento y cómo utilizarlo de manera adecuada.

Shaders

Un *shader* se puede definir como un conjunto de instrucciones software que permiten aplicar efectos de renderizado a primitivas geométricas. Al ejecutarse en las unidades de procesamiento gráfico (*Graphic Processing Units - GPUs*), el rendimiento de la aplicación gráfica mejora considerablemente.

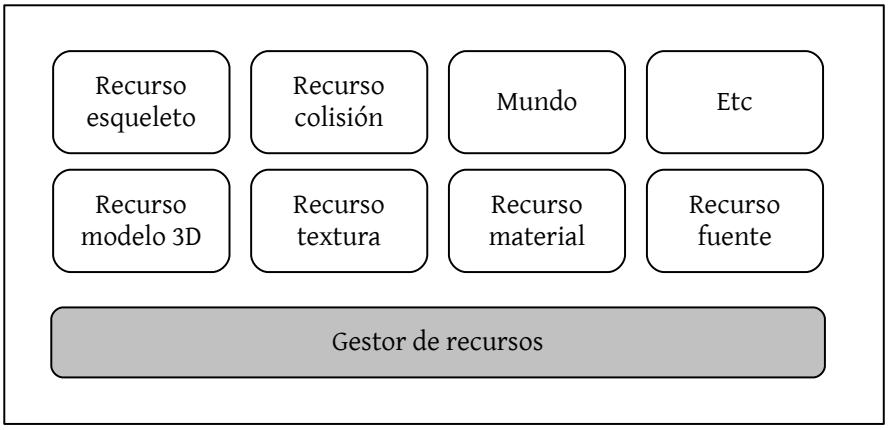


Figura 1.10: Visión conceptual del gestor de recursos y sus entidades asociadas. Esquema adaptado de la arquitectura propuesta en [42].

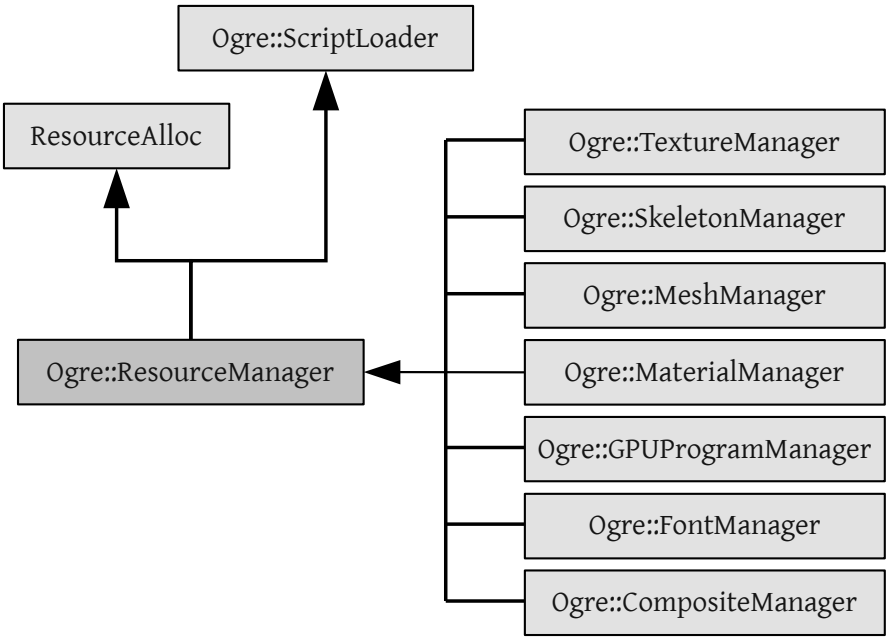


Figura 1.11: Diagrama de clases asociado al gestor de recursos de *Ogre 3D*, representado por la clase *Ogre::ResourceManager*.

1.2.6. Motor de *rendering*

Debido a que el componente gráfico es una parte fundamental de cualquier juego, junto con la necesidad de mejorarlo continuamente, el motor de renderizado es una de las partes más complejas de cualquier motor de juego.

Al igual que ocurre con la propia arquitectura de un motor de juegos, el enfoque más utilizado para diseñar el motor de renderizado consiste en utilizar una arquitectura multi-capa, como se puede apreciar en la figura 1.12.

A continuación se describen los principales módulos que forman parte de cada una de las capas de este componente.

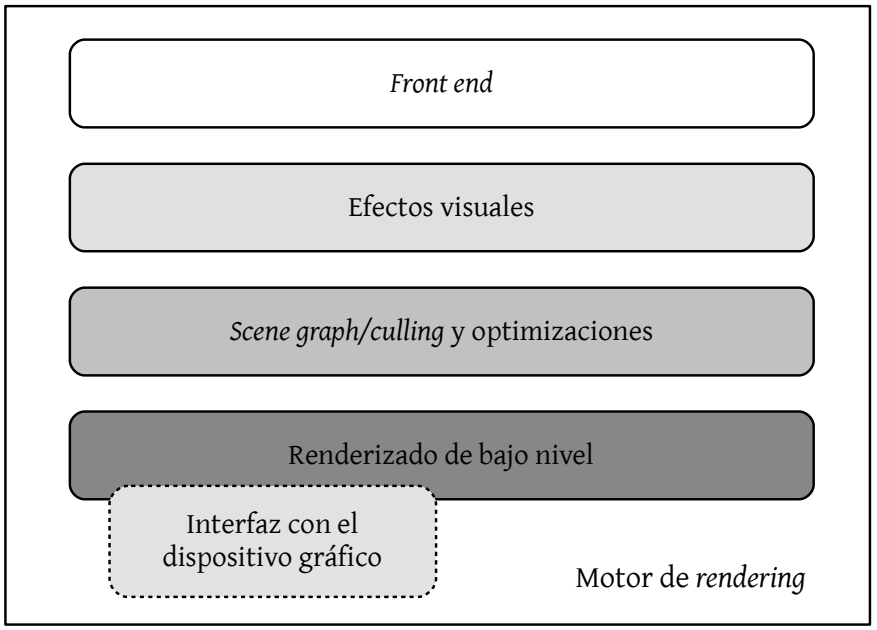


Figura 1.12: Visión conceptual de la arquitectura general de un motor de rendering. Esquema simplificado de la arquitectura discutida en [42].

La capa de **renderizado de bajo nivel** aglutina las distintas utilidades de renderizado del motor, es decir, la funcionalidad asociada a la representación gráfica de las distintas entidades que participan en un determinado entorno, como por ejemplo cámaras, primitivas de *rendering*, materiales, texturas, etc. El objetivo principal de esta capa reside precisamente en renderizar las distintas primitivas geométricas tan rápido como sea posible, sin tener en cuenta posibles optimizaciones ni considerar, por ejemplo, qué partes de las escenas son visibles desde el punto de vista de la cámara.

Esta capa también es responsable de gestionar la interacción con las APIs de programación gráficas, como *OpenGL* o *Direct3D*, simplemente para poder acceder a los distintos dispositivos gráficos que estén disponibles. Típicamente, este módulo se denomina *interfaz de dispositivo gráfico* (*graphics device interface*).

Así mismo, en la capa de renderizado de bajo nivel existen otros componentes encargados de procesar el dibujado de distintas primitivas geométricas, así como de la gestión de la cámara y los diferentes modos de proyección. En otras palabras, esta capa proporciona una serie de abstracciones para manejar tanto las primitivas geométricas como las cámaras virtuales y las propiedades vinculadas a las mismas.

Optimización

Las optimizaciones son esenciales en el desarrollo de aplicaciones gráficas, en general, y de videojuegos, en particular, para mejorar el rendimiento. Los desarrolladores suelen hacer uso de estructuras de datos auxiliares para aprovecharse del mayor conocimiento disponible sobre la propia aplicación.

Por otra parte, dicha capa también gestiona el estado del hardware gráfico y los *shaders* asociados. Básicamente, cada primitiva recibida por esta capa tiene asociado un material y se ve afectada por diversas fuentes de luz. Así mismo, el material describe la textura o texturas utilizadas por la primitiva y otras cuestiones como por ejemplo qué *pixel* y *vertex shaders* se utilizarán para renderizarla.

La capa superior a la de renderizado de bajo nivel se denomina ***scene graph/-culling* y optimizaciones** y, desde un punto de vista general, es la responsable de seleccionar qué parte o partes de la escena se enviarán a la capa de *rendering*. Esta selección, u optimización, permite incrementar el rendimiento del motor de *rendering*, debido a que se limita el número de primitivas geométricas enviadas a la capa de nivel inferior.

Aunque en la capa de *rendering* sólo se dibujan las primitivas que están dentro del campo de visión de la cámara, es decir, dentro del *viewport*, es posible aplicar más optimizaciones que simplifiquen la complejidad de la escena a renderizar, obviando aquellas partes de la misma que no son visibles desde la cámara. Este tipo de optimizaciones son críticas en juegos que tenga una complejidad significativa con el objetivo de obtener tasas de *frames* por segundo aceptables.

Una de las optimizaciones típicas consiste en hacer uso de estructuras de datos de *subdivisión espacial* para hacer más eficiente el renderizado, gracias a que es posible determinar de una manera rápida el conjunto de objetos potencialmente visibles. Dichas estructuras de datos suelen ser árboles, aunque también es posible utilizar otras alternativas. Tradicionalmente, las subdivisiones espaciales se conocen como *scene graph* (grafo de escena), aunque en realidad representan un caso particular de estructura de datos.

Por otra parte, en esta capa también es común integrar métodos de *culling*, como por ejemplo aquellos basados en utilizar información relevante de las oclusiones para determinar qué objetos están siendo solapados por otros, evitando que los primeros se tengan que enviar a la capa de *rendering* y optimizando así este proceso.

Idealmente, esta capa debería ser independiente de la capa de renderizado, permitiendo así aplicar distintas optimizaciones y abstrayéndose de la funcionalidad relativa al dibujado de primitivas. Un ejemplo representativo de esta independencia está representado por OGRE (Object-Oriented Graphics Rendering Engine) y el uso de la filosofía *plug & play*, de manera que el desarrollador puede elegir distintos diseños de grafos de escenas ya implementados y utilizarlos en su desarrollo.

Filosofía *Plug & Play*

Esta filosofía se basa en hacer uso de un componente funcional, hardware o software, sin necesidad de configurar ni de modificar el funcionamiento de otros componentes asociados al primero.

Sobre la capa relativa a las optimizaciones se sitúa la capa de **efectos visuales**, la cual proporciona soporte a distintos efectos que, posteriormente, se puedan integrar en los juegos desarrollados haciendo uso del motor. Ejemplos representativos de módulos que se incluyen en esta capa son aquéllos responsables de gestionar los sistemas de partículos (humo, agua, etc), los mapeados de entorno o las sombras dinámicas.

Finalmente, la capa de ***front-end*** suele estar vinculada a funcionalidad relativa a la superposición de contenido 2D sobre el escenario 3D. Por ejemplo, es bastante común utilizar algún tipo de módulo que permita visualizar el menú de un juego o la interfaz gráfica que permite conocer el estado del personaje principal del videojuego (inventario, armas, herramientas, etc). En esta capa también se incluyen componentes para reproducir vídeos previamente grabados y para integrar secuencias cinemáticas, a veces interactivas, en el propio videojuego. Este último componente se conoce como IGC (In-Game Cinematics) *system*.

1.2.7. Herramientas de depuración

Debido a la naturaleza intrínseca de un videojuego, vinculada a las aplicaciones gráficas en tiempo real, resulta esencial contar con buenas herramientas que permitan depurar y optimizar el propio motor de juegos para obtener el mejor rendimiento posible. En este contexto, existe un gran número de herramientas de este tipo. Algunas de ellas son herramientas de propósito general que se pueden utilizar de manera externa al motor de juegos. Sin embargo, la práctica más habitual consiste en construir herramientas de *profiling*, vinculadas al análisis del rendimiento, o depuración que estén asociadas al propio motor. Algunas de las más relevantes se enumeran a continuación [42]:

Versiones *beta*

Además del uso extensivo de herramientas de depuración, las desarrolladoras de videojuegos suelen liberar versiones betas de los mismos para que los propios usuarios contribuyan en la detección de *bugs*.

- Mecanismos para determinar el tiempo empleado en ejecutar un fragmento específico de código.
- Utilidades para mostrar de manera gráfica el rendimiento del motor mientras se ejecuta el juego.
- Utilidades para volcar *logs* en ficheros de texto o similares.
- Herramientas para determinar la cantidad de memoria utilizada por el motor en general y cada subsistema en particular. Este tipo de herramientas suelen tener distintas vistas gráficas para visualizar la información obtenida.
- Herramientas de depuración que gestión el nivel de información generada.
- Utilidades para grabar eventos particulares del juego, permitiendo reproducirlos posteriormente para depurar *bugs*.

1.2.8. Motor de física

La detección de colisiones en un videojuego y su posterior tratamiento resultan esenciales para dotar de realismo al mismo. Sin un mecanismo de detección de colisiones, los objetos se *traspasarían* unos a otros y no sería posible interactuar con ellos. Un ejemplo típico de colisión está representado en los juegos de conducción por el choque entre dos o más vehículos. Desde un punto de vista general, el sistema de detección de colisiones es responsable de llevar a cabo las siguientes tareas [6]:

ED auxiliares

Al igual que ocurre en procesos como la obtención de la posición de un enemigo en el mapa, el uso extensivo de estructuras de datos auxiliares permite obtener soluciones a problemas computacionalmente complejos. La gestión de colisiones es otro proceso que se beneficia de este tipo de técnicas.

1. La **detección de colisiones**, cuya salida es un valor lógico indicando si hay o no colisión.
2. La **determinación de la colisión**, cuya tarea consiste en calcular el punto de intersección de la colisión.
3. La **respuesta a la colisión**, que tiene como objetivo determinar las acciones que se generarán como consecuencia de la misma.

Debido a las restricciones impuestas por la naturaleza de tiempo real de un videojuego, los mecanismos de gestión de colisiones se suelen aproximar para simplificar la complejidad de los mismos y no reducir el rendimiento del motor. Por ejemplo, en algunas ocasiones los objetos 3D se aproximan con una serie de líneas, utilizando técnicas de intersección de líneas para determinar la existancia o no de una colisión. También es bastante común hacer uso de árboles BSP para representar el entorno y optimizar la detección de colisiones con respecto a los propios objetos.

Por otra parte, algunos juegos incluyen sistemas realistas o semi-realistas de simulación dinámica. En el ámbito de la industria del videojuego, estos sistemas se suelen denominar ***sistema de física*** y están directamente ligados al sistema de gestión de colisiones.

Actualmente, la mayoría de compañías utilizan motores de colisión/física desarrollados por terceras partes, integrando estos *kits* de desarrollo en el propio motor. Los más conocidos en el ámbito comercial son *Havok*, el cual representa el estándar de facto en la industria debido a su potencia y rendimiento, y *PhysX*, desarrollado por *NVIDIA* e integrado en motores como por ejemplo el *Unreal Engine 3*.

En el ámbito del *open source*, uno de los más utilizados es ODE. Sin embargo, en este curso se hará uso del motor de simulación física *Bullet*[14], el cual se utiliza actualmente en proyectos tan ambiciosos como la suite 3D *Blender*.

1.2.9. Interfaces de usuario

En cualquier tipo de juego es necesario desarrollar un módulo que ofrezca una abstracción respecto a la interacción del usuario, es decir, un módulo que principalmente sea responsable de procesar los **eventos de entrada** del usuario. Típicamente, dichos eventos estarán asociados a la pulsación de una tecla, al movimiento del ratón o al uso de un *joystick*, entre otros.

Desde un punto de vista más general, el módulo de interfaces de usuario también es responsable del **tratamiento de los eventos** de salida, es decir, aquellos eventos que proporcionan una retroalimentación al usuario. Dicha interacción puede estar representada, por ejemplo, por el sistema de vibración del mando de una consola o por la fuerza ejercida por un volante que está siendo utilizado en un juego de conducción. Debido a que este módulo gestiona los eventos de entrada y de salida, se suele denominar comúnmente *componente de entrada/salida del jugador* (*player I/O component*).

El módulo de interfaces de usuario actúa como un puente entre los detalles de bajo nivel del hardware utilizado para interactuar con el juego y el resto de controles de más alto nivel. Este módulo también es responsable de otras tareas importantes, como la asocación de acciones o funciones lógicas al sistema de control del juego, es decir, permite asociar eventos de entrada a acciones lógicas de alto nivel.

En la gestión de eventos se suelen utilizar patrones de diseño como el patrón *delegate* [37], de manera que cuando se detecta un evento, éste se traslada a la entidad adecuada para llevar a cabo su tratamiento.

1.2.10. *Networking* y multijugador

Lag

El retraso que se produce desde que se envía un paquete de datos por una entidad hasta que otra lo recibe se conoce como *lag*. En el ámbito de los videojuegos, el *lag* se suele medir en milésimas de segundo.

La mayoría de juegos comerciales desarrollados en la actualidad incluyen modos de juegos multijugador, con el objetivo de incrementar la jugabilidad y duración de los títulos lanzados al mercado. De hecho, algunas compañías basan el modelo de negocio de algunos de sus juegos en el **modo *online***, como por ejemplo *World of Warcraft* de *Blizzard Entertainment*, mientras algunos títulos son ampliamente conocidos por su exitoso modo multijugador *online*, como por ejemplo la saga *Call of Duty* de *Activision*.

Aunque el modo multijugador de un juego puede resultar muy parecido a su versión *single-player*, en la práctica incluir el soporte de varios jugadores, ya sea *online* o no, tiene un profundo impacto en diseño de ciertos componentes del motor de juego, como por ejemplo el modelo de objetos del juego, el motor de renderizado, el módulo

[14] `http://www.bulletphysics.com`

de entrada/salida o el sistema de animación de personajes, entre otros. De hecho, una de las filosofías más utilizadas en el diseño y desarrollo de motores de juegos actuales consiste en tratar el modo de un único jugador como un caso particular del modo multijugador.

Por otra parte, el **módulo de *networking*** es el responsable de *informar* de la evolución del juego a los distintos actores o usuarios involucrados en el mismo mediante el envío de paquetes de información. Típicamente, dicha información se transmite utilizando *sockets*. Con el objetivo de reducir la latencia del modo multijugador, especialmente a través de Internet, sólo se envía/recibe información relevante para el correcto funcionamiento de un juego. Por ejemplo, en el caso de los FPS, dicha información incluye típicamente la posición de los jugadores en cada momento, entre otros elementos.

1.2.11. Subsistema de juego

El subsistema de juego, conocido por su término en inglés *gameplay*, integra todos aquellos módulos relativos al funcionamiento interno del juego, es decir, aglutina tanto las propiedades del mundo virtual como las de los distintos personajes. Por una parte, este subsistema permite la definición de las reglas que gobiernan el mundo virtual en el que se desarrolla el juego, como por ejemplo la necesidad de derrotar a un enemigo antes de enfrentarse a otro de mayor nivel. Por otra parte, este subsistema también permite la definición de la mecánica del personaje, así como sus objetivos durante el juego.

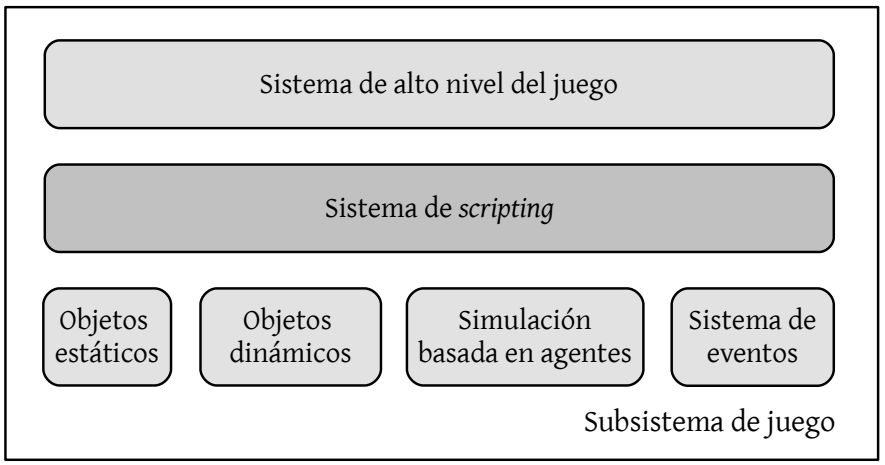


Figura 1.13: Visión conceptual de la arquitectura general del subsistema de juego. Esquema simplificado de la arquitectura discutida en [42].

Este subsistema sirve también como capa de *aislamiento* entre las capas de más bajo nivel, como por ejemplo la de *rendering*, y el propio funcionamiento del juego. Es decir, uno de los principales objetivos de diseño que se persiguen consiste en independizar la lógica del juego de la implementación subyacente. Por ello, en esta capa es bastante común encontrar algún tipo de sistema de ***scripting*** o lenguaje de alto nivel para definir, por ejemplo, el comportamiento de los personajes que participan en el juego.

Diseñando juegos

Los diseñadores de los niveles de un juego, e incluso del comportamiento de los personajes y los NPCs, suelen dominar perfectamente los lenguajes de script, ya que son su principal herramienta para llevar a cabo su tarea.

La capa relativa al subsistema de juego maneja conceptos como el *mundo del juego*, el cual se refiere a los distintos elementos que forman parte del mismo, ya sean estáticos o dinámicos. Los tipos de objetos que forman parte de ese mundo se suelen denominar *modelo de objetos del juego* [42]. Este modelo proporciona una simulación en tiempo real de esta colección heterogénea, incluyendo

- Elementos geométricos relativos a fondos estáticos, como por ejemplo edificios o carreteras.
- Cuerpos rígidos dinámicos, como por ejemplo rocas o sillas.
- El propio personaje principal.
- Los personajes no controlados por el usuario (NPCs).
- Cámaras y luces virtuales.
- Armas, proyectiles, vehículos, etc.

El modelo de objetos del juego está intimamente ligado al *modelo de objetos software* y se puede entender como el conjunto de propiedades del lenguaje, políticas y convenciones utilizadas para implementar código utilizando una filosofía de orientación a objetos. Así mismo, este modelo está vinculado a cuestiones como el lenguaje de programación empleado o a la adopción de una política basada en el uso de patrones de diseño, entre otras.

En la capa de subsistema de juego se integra el **sistema de eventos**, cuya principal responsabilidad es la de dar soporte a la comunicación entre objetos, independientemente de su naturaleza y tipo. Un enfoque típico en el mundo de los videojuegos consiste en utilizar una *arquitectura dirigida por eventos*, en la que la principal entidad es el evento. Dicho evento consiste en una estructura de datos que contiene información relevante de manera que la comunicación está precisamente guiada por el contenido del evento, y no por el emisor o el receptor del mismo. Los objetos suelen implementar manejadores de eventos (*event handlers*) para tratarlos y actuar en consecuencia.

Por otra parte, el **sistema de *scripting*** permite modelar fácilmente la lógica del juego, como por ejemplo el comportamiento de los enemigos o NPCs, sin necesidad de volver a compilar para comprobar si dicho comportamiento es correcto o no. En algunos casos, los motores de juego pueden seguir en funcionamiento al mismo tiempo que se carga un nuevo *script*.

Finalmente, en la capa del subsistema de juego es posible encontrar algún módulo que proporcione funcionalidad añadida respecto al tratamiento de la IA, normalmente de los NPCs. Este tipo de módulos, cuya funcionalidad se suele incluir en la propia capa de software específica del juego en lugar de integrarla en el propio motor, son cada vez más populares y permiten asignar comportamientos preestablecidos sin necesidad de programarlos. En este contexto, la *simulación basada en agentes* [102] cobra especial relevancia.

Este tipo de módulos pueden incluir aspectos relativos a problemas clásicos de la IA, como por ejemplo la búsqueda de caminos óptimos entre dos puntos, conocida como *pathfinding*, y típicamente vinculada al uso de algoritmos $A*$ [79]. Así mismo, también es posible hacer uso de información *privilegiada* para optimizar ciertas tareas, como por ejemplo la localización de entidades de interés para agilizar el cálculo de aspectos como la detección de colisiones.

I'm all ears!

El apartado sonoro de un juego es especialmente importante para que el usuario se sienta inmerso en el mismo y es crítico para acompañar de manera adecuada el desarrollo de dicho juego.

1.2.12. Audio

Tradicionalmente, el mundo del desarrollo de videojuegos siempre ha prestado más atención al componente gráfico. Sin embargo, el apartado sonoro también tiene una gran importancia para conseguir una inmersión total del usuario en el juego. Por ello, el **motor de audio** ha ido cobrando más y más relevancia.

Asimismo, la aparición de nuevos formatos de audio de alta definición y la popularidad de los sistemas de cine en casa han contribuido a esta evolución en el cada vez más relevante apartado sonoro.

Actualmente, al igual que ocurre con otros componentes de la arquitectura del motor de juego, es bastante común encontrar desarrollos listos para utilizarse e integrarse en el motor de juego, los cuales han sido realizados por compañías externas a la del propio motor. No obstante, el apartado sonoro también requiere modificaciones que son específicas para el juego en cuestión, con el objetivo de obtener un alto de grado de fidelidad y garantizar una buena experiencia desde el punto de visto auditivo.

1.2.13. Subsistemas específicos de juego

Por encima de la capa de subsistema de juego y otros componentes de más bajo nivel se sitúa la capa de subsistemas específicos de juego, en la que se integran aquellos módulos responsables de ofrecer las características propias del juego. En función del tipo de juego a desarrollar, en esta capa se situarán un mayor o menor número de módulos, como por ejemplo los relativos al sistema de cámaras virtuales, mecanismos de IA específicos de los personajes no controlados por el usuario (NPCs), aspectos de renderizados específicos del juego, sistemas de armas, puzzles, etc.

Idealmente, la línea que separa el motor de juego y el propio juego en cuestión estaría entre la capa de subsistema de juego y la capa de subsistemas específicos de juego.

Capítulo 2

Herramientas de Desarrollo

Cleto Martín Angelina

Actualmente, existen un gran número de aplicaciones y herramientas que permiten a los desarrolladores de videojuegos, y de aplicaciones en general, aumentar su productividad a la hora de construir software, gestionar los proyectos y recursos, así como automatizar procesos de construcción.

En este capítulo, se pone de manifiesto la importancia de la gestión en un proyecto software y se muestran algunas de las herramientas de desarrollo más conocidas en sistemas GNU/Linux. La elección de este tipo de sistema no es casual. Por un lado, se trata de Software Libre, lo que permite a desarrolladores estudiar, aprender y entender lo que hace el código que se ejecuta. Por otro lado, probablemente sea el mejor sistema operativo para construir y desarrollar aplicaciones debido al gran número de herramientas que proporciona. Es un sistema hecho por programadores para programadores.

2.1. Introducción

En la construcción de software no trivial, las herramientas de gestión de proyectos y de desarrollo facilitan la labor de las personas que lo construyen. Conforme el software se va haciendo más complejo y se espera más funcionalidad de él, se hace necesario el uso de herramientas que permitan automatizar los procesos del desarrollo, así como la gestión del proyecto y su documentación.

Además, dependiendo del contexto, es posible que existan otros integrantes del proyecto que no tengan formación técnica y que necesiten realizar labores sobre el producto como traducciones, pruebas, diseño gráfico, etc.

Los videojuegos son proyectos software que, normalmente, requieren la participación de varias personas con diferentes perfiles profesionales (programadores, diseñadores gráficos, compositores de sonidos, etc.). Cada uno de ellos, trabaja con diferentes tipos de datos en diferentes tipos de formatos. Todas las herramientas que permitan la construcción automática del proyecto, la integración de sus diferentes componentes y la coordinación de sus miembros serán de gran ayuda en un entorno tan heterogéneo.

Desde el punto de vista de la gestión del proyecto, una tarea esencial es la automatización del proceso de compilación y de construcción de los programas. Una de las tareas que más frecuentemente se realizan mientras se desarrolla y depura un programa es la de compilación y construcción. Cuanto más grande y complejo sea un programa, mayor es el tiempo que se pierde en esta fase. Por tanto, un proceso automático de construcción de software ahorrará muchas pérdidas de tiempo en el futuro.

En los sistemas GNU/Linux es habitual el uso de herramientas como el compilador GCC, el sistema de construcción GNU Make y el depurador GDB. Todas ellas creadas en el proyecto GNU y orientadas a la creación de programas en C/C++, aunque también pueden ser utilizadas con otras tecnologías. También existen editores de texto como GNU Emacs o vi, y modernos (pero no por ello mejores) entornos de desarrollo como Eclipse que no sólo facilitan las labores de escritura de código, sino que proporcionan numerosas herramientas auxiliares dependiendo del tipo de proyecto. Por ejemplo, Eclipse puede generar los archivos Makefiles necesarios para automatizar el proceso de construcción con GNU Make.

Figura 2.1: El proyecto GNU proporciona una gran abanico de herramientas de desarrollo y son utilizados en proyectos software de todo tipo.

2.2. Compilación, enlazado y depuración

La compilación y la depuración y, en general, el proceso de construcción es una de las tareas más importantes desde el punto de vista del desarrollador de aplicaciones escritas en C/C++. En muchas ocasiones, parte de los problemas en el desarrollo de un programa vienen originados directa o indirectamente por el propio proceso de construcción del mismo. Hacer un uso indebido de las opciones del compilador, no depurar utilizando los programas adecuados o realizar un incorrecto proceso de construcción del proyecto son ejemplos típicos que, en muchas ocasiones, consumen demasiado tiempo en el desarrollo. Por todo ello, tener un conocimiento sólido e invertir tiempo en estas cuestiones ahorra más de un quebradero de cabeza a lo largo del ciclo de vida de la aplicación.

En esta sección se estudia una terminología y conceptos básicos en el ámbito de los procesos de construcción de aplicaciones.

Concretamente, se muestra el uso del compilador de C/C++ GCC, el depurador GDB y el sistema de construcción automático GNU Make.

Figura 2.2: GCC es una colección de compiladores para lenguajes como C/C++ y Java.

2.2.1. Conceptos básicos

A la hora de buscar y compartir información con el resto de compañeros de profesión es necesario el uso de una terminología común. En las tareas de construcción del software existen algunos términos que son importantes conocer.

Código fuente, código objeto y código ejecutable

Como es de suponer, la programación consiste en escribir programas. Los **programas** son procedimientos que, al ejecutarse de forma secuencial, se obtienen unos resultados. En muchos sentidos, un programa es como una receta de cocina: una especificación secuencial de las acciones que hay que realizar para conseguir un objetivo. Cómo de abstractas sean estas especificaciones es lo que define el **nivel de abstracción** de un lenguaje.

Los programas se pueden escribir directamente en **código ejecutable**, también llamado código binario o código máquina. Sin embargo, el nivel de abstracción tan bajo que ofrecen estos lenguajes haría imposible que muchos proyectos actuales pudieran llevarse a cabo. Este código es el que entiende la máquina donde se va a ejecutar el programa y es específico de la plataforma. Por ejemplo, máquinas basadas en la arquitectura PC no ofrecen el mismo **repertorio de instrucciones** que otras basadas en la arquitectura PPC o ARM. A la dificultad de escribir código de bajo nivel se le suma la característica de no ser portable.

Por este motivo se han creado los **compiladores**. Estos programas traducen **código fuente**, programado en un lenguaje de alto nivel, en el código ejecutable para una plataforma determinada. Un paso intermedio en este proceso de compilación es la generación de **código objeto**, que no es sino código en lenguaje máquina al que le falta realizar el *proceso de enlazado*.

Aunque en los sistemas como GNU/Linux la extensión en el nombre de los archivos es puramente informativa, los archivos fuente en C++ suelen tener las extensiones `.cpp`, `.cc`, y `.h`, `.hh` o `.hpp` para las cabeceras. Por su parte, los archivos de código objeto tienen extensión `.o` y lo ejecutables no suelen tener extensión.

Compilador

Como ya se ha dicho, se trata de un programa que, a partir del código fuente, genera el código ejecutable para la máquina destino. Este proceso de *traducción* automatizado permite al programador, entre otras muchas ventajas:

- No escribir código de muy bajo nivel.
- Abstraerse de la características propias de la máquina tales como registros especiales, modos de acceso a memoria, etc.
- Escribir código portable. Basta con que exista un compilador en una plataforma que soporte C++ para que un programa pueda ser portado.

Aunque la función principal del compilador es la de actuar como **traductor** entre dos tipos de lenguaje, este término se reserva a los programas que transforman de un lenguaje de alto nivel a otro; por ejemplo, el programa que transforma código C++ en Java.

Existen muchos compiladores comerciales de C++ como Borland C++, Microsoft Visual C++. Sin embargo, GCC es un compilador libre y gratuito que soporta C/C++ (entre otros lenguajes) y es ampliamente utilizado en muchos sectores de la informática: desde la programación gráfica a los sistemas empotrados.

Obviamente, las diferencias entre las implementaciones vienen determinadas por el contexto de aplicación para los que son concebidos. Sin embargo, es posible extraer una estructura funcional común como muestra la figura 2.3, que representa las **fases de compilación** en las que, normalmente, está dividido el proceso de compilación.

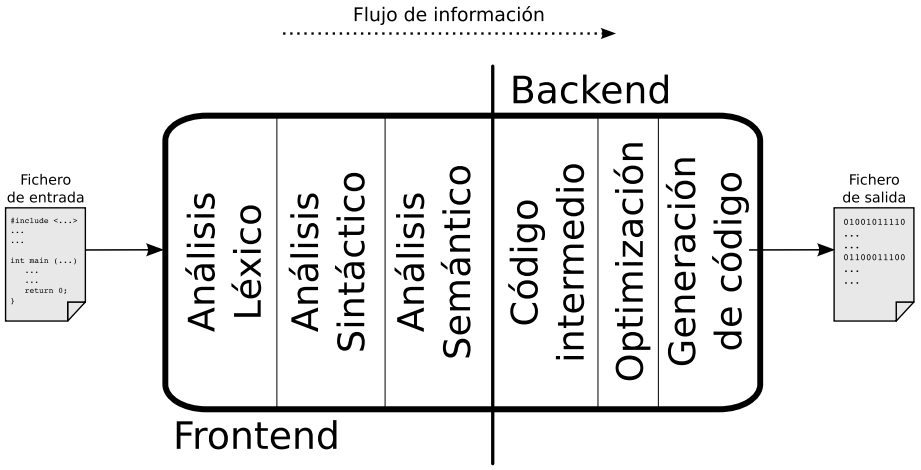


Figura 2.3: Fases de compilación

Las fases están divididas en dos grandes bloques:

- **Frontend**: o frontal del compilador. Es el encargado de realizar el análisis léxico, sintáctico y semántico de los ficheros de entrada. El resultado de esta fase es un **código intermedio** que es independiente de la plataforma destino.
- **Backend**: el código intermedio pasa por el **optimizador** y es mejorado utilizando diferentes estrategias como la eliminación de código muerto o de situaciones redundantes. Finalmente, el código intermedio optimizado lo toma un **generador** de código máquina específico de la plataforma destino. Además de la generación, en esta fase también se realizan algunas optimizaciones propias de la plataforma destino.

En definitiva, el proceso de compilación de un compilador está dividido en etapas bien diferenciadas, que proporcionan diferente funcionalidad para las etapas siguientes hasta la generación final.

Enlazador

Un programa puede estar compuesto por varios **módulos**, lo cual permite que un proyecto pueda ser más mantenible y manejable. Los módulos pueden tener independencias entre sí y la comprobación y resolución de estas dependencias corren a cargo del **enlazador**. El enlazador toma como entrada el código objeto.

Bibliotecas

Una de las principales ventajas del software es la reutilización del código. Normalmente, los problemas pueden resolverse utilizando código ya escrito anteriormente y la reutilización del mismo se vuelve un aspecto clave para el tiempo de desarrollo del producto. Las **bibliotecas** ofrecen una determinada funcionalidad ya implementada para que sea utilizada por programas. Las bibliotecas se incorporan a los programas durante el proceso de enlazado.

Las bibliotecas pueden *enlazarse* contra el programa de dos formas:

- **Estáticamente**: en tiempo de enlazado, se resuelven todas las dependencias y símbolos que queden por definir y se incorpora al ejecutable final.

  La principal ventaja de utilizar enlazado estático es que el ejecutable puede considerarse *standalone* y es completamente independiente. El sistema donde vaya a ser ejecutado no necesita tener instaladas bibliotecas externas de antemano. Sin embargo, el código ejecutable generado tiene mayor tamaño.

- **Dinámicamente**: en tiempo de enlazado sólo se comprueba que ciertas dependencias y símbolos estén definidos, pero no se incorpora al ejecutable. Será en tiempo de ejecución cuando se realizará la carga de la biblioteca en memoria.

  El código ejecutable generado es mucho menor, pero el sistema debe tener la biblioteca previamente instalada.

En sistemas GNU/Linux, las bibliotecas ya compiladas suelen encontrarse en `/usr/lib` y siguen un convenio para el nombre: `lib`*nombre*. Las bibliotecas dinámicas tienen extensión `.so` y las estáticas `.a`.

2.2.2. Compilando con GCC

Desde un punto de vista estricto, GCC no es un compilador. GNU Compiler Collection (GCC) es un conjunto de compiladores que proporciona el proyecto GNU para diferentes lenguajes de programación tales como C, C++, Java, FORTRAN, etc. Dentro de este conjunto de compiladores, G++ es el compilador para C++. Teniendo en cuenta esta precisión, es común llamar simplemente GCC al compilador de C y C++, por lo que se usarán de forma indistinta en este texto.

En esta sección se introducen la estructura básica del compilador GCC, así como algunos de sus componentes y su papel dentro del proceso de compilación. Finalmente, se muestran ejemplos de cómo utilizar GCC (y otras herramientas auxiliares) para construir un ejecutable, una biblioteca estática y una dinámica.

2.2.3. ¿Cómo funciona GCC?

GCC es un compilador cuya estructura es muy similar a la presentada en la sección 2.2.1. Sin embargo, cada una de las fases de compilación la realiza un componente bien definido e independiente. Concretamente, al principio de la fase de compilación, se realiza un procesamiento inicial del código fuente utilizando el *preprocesador* GNU CPP, posteriormente se utiliza GNU Assembler para obtener el código objeto y, con la ayuda del *enlazador* GNU ld, se crea el binario final.

En la figura 2.4 se muestra un esquema general de los componentes de GCC que participan en el proceso.

El hecho de que esté dividido en estas etapas permite una compilación **modular**, es decir, cada fichero de entrada se transforma a código objeto y con la ayuda del enlazador se resuelven las dependencias que puedan existir entre ellos. A continuación, se comenta brevemente cada uno de los componentes principales.

distcc

La compilación modular y por fases permite que herramientas como `distcc` puedan realizar compilaciones distribuidas en red y en paralelo.

Preprocesador

El preprocesamiento es la primera transformación que sufre un programa en C/C++. Se lleva a cabo por el GNU CPP y, entre otras, realiza las siguientes tareas:

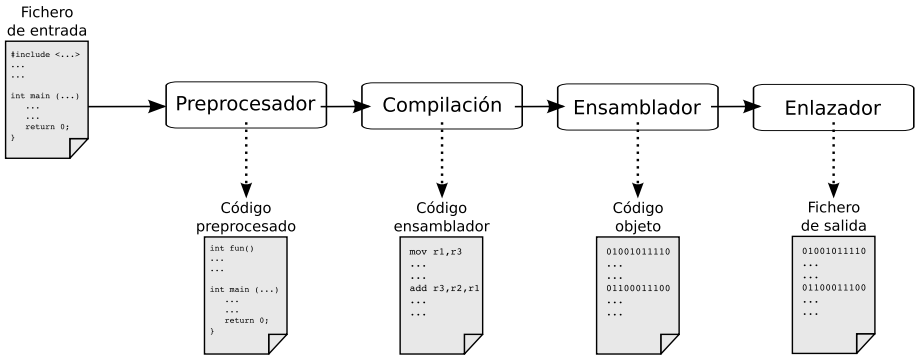


Figura 2.4: Proceso de compilación en GCC

- Inclusión efectiva del código incluido por la directiva `#include`.
- Resuelve de las directivas `#ifdef`/`#ifndef` para la compilación condicional.
- Sustitución efectiva de todas las directivas de tipo `#define`.

El preprocesador se puede invocar directamente utilizando la orden `cpp`. Como ejercicio se reserva al lector observar qué ocurre al invocar al preprocesador con el siguiente fragmento de código. ¿Se realizan comprobaciones lexícas, sintáticas o semánticas?. Utilice los parámetros que ofrece el programa para definir la macro `DEFINED_IT`.

Listado 2.1: Código de ejemplo preprocesable

```
1  #include <iostream>
2  #define SAY_HELLO "Hi, world!"
3
4  #ifdef DEFINED_IT
5  #warning "If you see this message, you DEFINED_IT"
6  #endif
7
8  using namespace std;
9
10 Code here??
11
12 int main() {
13   cout << SAY_HELLO << endl;
14   return 0;
15 }
```

Compilación

El código fuente, una vez preprocesado, se compila a lenguaje ensamblador, es decir, a una representación de bajo nivel del código fuente. Originalmente, la sintaxis de este lenguaje es la de AT&T pero desde algunas versiones recientes también se soporta la sintaxis de Intel.

Entre otras muchas operaciones, en la compilación se realizan las siguientes operaciones:

- Análisis **sintáctico** y **semántico** del programa. Pueden ser configurados para obtener diferentes mensajes de advertencia (*warnings*) a diferentes niveles.

- Comprobación y resolución de símbolos y dependencias a nivel de **declaración**.
- Realizar optimizaciones.

Utilizando GCC con la opción `-S` puede detenerse el proceso de compilación hasta la generación del código ensamblador. Como ejercicio, se propone cambiar el código fuente anterior de forma que se pueda construir el correspondiente en ensamblador.

GCC proporciona diferentes niveles de optimizaciones (opción `-O`). Cuanto mayor es el nivel de optimización del código resultante, mayor es el tiempo de compilación pero suele hacer más eficiente el código de salida. Por ello, se recomienda **no optimizar** el código durante las fases de desarrollo y sólo hacerlo en la fase de distribución/instalación del software.

Ensamblador

Una vez se ha obtenido el código ensamblador, GNU Assembler es el encargado de realizar la traducción a **código objeto** de cada uno de los módulos del programa. Por defecto, el código objeto se genera en archivos con extensión `.o` y la opción `-c` de GCC permite detener el proceso de compilación en este punto.

GNU Assembler

GNU Assembler forma parte de la distribución GNU Binutils y se corresponde con el programa `as`.

Como ejercicio, se propone al lector modificar el código ensamblador obtenido en la fase anterior sustituyendo el mensaje original `"Hi, world"` por `"Hola, mundo"`. Generar el código objeto asociado utilizando directamente el ensamblador (no GCC).

Enlazador

GNU Linker

GNU Linker también forma parte de la distribución GNU Binutils y se corresponde con el programa `ld`.

Con todos los archivos objetos el **enlazador** (*linker*) es capaz de generar el ejecutable o código binario final. Algunas de las tareas que se realizan en el proceso de enlazado son las siguientes:

- Selección y filtrado de los objetos necesarios para la generación del binario.
- Comprobación y resolución de símbolos y dependencias a nivel de **definición**.
- Realización del enlazado (estático y dinámico) de las bibliotecas.

Como ejercicio se propone utilizar el *linker* directamente con el código objeto generado en el apartado anterior. Nótese que las opciones `-l` y `-L` sirven para añadir rutas personalizadas a las que por defecto `ld` utiliza para buscar bibliotecas.

2.2.4. Ejemplos

Como se ha mostrado, el proceso de compilación está compuesto por varias fases bien diferenciadas. Sin embargo, con GCC se integra todo este proceso de forma que, a partir del código fuente se genere el binario final.

En esta sección se mostrarán ejemplos en los que se crea un ejecutable al que, posteriormente, se enlaza con una biblioteca estática y otra dinámica.

Compilación de un ejecutable

Como ejemplo de ejecutable se toma el siguiente programa.

Listado 2.2: Programa básico de ejemplo

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Square {
6  private:
7    int side_;
8
9  public:
10   Square(int side_length) : side_(side_length) { };
11   int getArea() const { return side_*side_; };
12 };
13
14 int main () {
15   Square square(5);
16   cout << "Area: " << square.getArea() << endl;
17   return 0;
18 }
```

En C++, los programas que generan un ejecutable deben tener definida la función `main`, que será el punto de entrada de la ejecución. El programa es trivial: se define una clase `Square` que representa a un cuadrado. Ésta implementa un método `getArea()` que devuelve el área del cuadrado.

Suponiendo que el archivo que contiene el código fuente se llama `main.cpp`, para construir el binario utilizaremos `g++`, el compilador de C++ que se incluye en GCC. Se podría utilizar `gcc` y que se seleccionara automáticamente el compilador. Sin embargo, es una buena práctica utilizar el compilador correcto:

```
$ g++ -o main main.cpp
```

Nótese que todo el proceso de compilación se ha realizado automáticamente y cada una de las herramientas auxiliares se han ejecutado internamente en su momento oportuno (preprocesamiento, compilación, ensamblado y enlazado).

La opción `-o` indica a GCC el nombre del archivo de salida de la compilación.

Compilación de un ejecutable (modular)

En un proyecto, lo natural es dividir el código fuente en módulos que realizan operaciones concretas y bien definidas. En el ejemplo, podemos considerar un módulo la declaración y definición de la clase **Square**. Esta extracción se puede realizar de muchas maneras. Lo habitual es crear un fichero de cabecera `.h` con la declaración de la clase y un fichero `.cpp` con la definición:

Listado 2.3: Archivo de cabecera `Square.h`

```
1 class Square {
2 private:
3   int side_;
4
5 public:
6   Square(int side_length);
7   int getArea() const;
8 };
```

Listado 2.4: Implementación (`Square.cpp`)

```
1 #include "Square.h"
2
3 Square::Square (int side_length) : side_(side_length)
4 { }
5
6 int
7 Square::getArea() const
8 {
9   return side_*side_;
10 }
```

De esta forma, el archivo `main.cpp` quedaría como sigue:

Listado 2.5: Programa principal

```
1 #include <iostream>
2 #include "Square.h"
3
4 using namespace std;
5
6 int main () {
7   Square square(5);
8   cout << "Area: " << square.getArea() << endl;
9   return 0;
10 }
```

Para construir el programa, se debe primero construir el código objeto del módulo y añadirlo a la compilación de la función principal `main`. Suponiendo que el archivo de cabecera se encuentra en un directorio llamado `headers`, la compilación puede realizarse de la siguiente manera:

```
$ g++ -Iheaders -c Square.cpp
$ g++ -Iheaders -c main.cpp
$ g++ Square.o main.o -o main
```

También se puede realizar todos los pasos al mismo tiempo:

```
$ g++ -Iheaders Square.cpp main.cpp -o main
```

Con la opción `-I`, que puede aparecer tantas veces como sea necesario, se puede añadir rutas donde se buscarán las cabeceras. Nótese que, por ejemplo, en `main.cpp` se incluyen las cabeceras usando los símbolos <> y . Se recomienda utilizar los primeros para el caso en que las cabeceras forman parte de una API pública (si existe) y deban ser utilizadas por otros programas. Por su parte, las comillas se suelen utilizar para cabeceras internas al proyecto. Las rutas por defecto son el directorio actual `.` para las cabeceras incluidas con y para el resto el directorio del sistema (normalmente, `/usr/include`).

Como norma general, una buena costumbre es generar todos los archivos de código objeto de un módulo y añadirlos a la compilación con el programa principal.

Compilación de una biblioteca estática

Para este ejemplo se supone que se pretende construir una biblioteca con la que se pueda enlazar estáticamente y que contiene una jerarquía de clases correspondientes a 3 tipos de figuras (`Figure`): `Square`, `Triangle` y `Circle`. Cada figura está implementada como un módulo (cabecera + implementación):

Listado 2.6: `Figure.h`

```
1 #ifndef FIGURE_H
2 #define FIGURE_H
3
4 class Figure {
5 public:
6   virtual float getArea() const = 0;
7 };
8
9 #endif
```

Listado 2.7: `Square.h`

```
1 #include <Figure.h>
2
3 class Square : public Figure {
4 private:
5   float side_;
6
7 public:
8   Square(float side_length);
9   float getArea() const;
10 };
```

Listado 2.8: `Square.cpp`

```
1 #include "Square.h"
2
3 Square::Square (float side) : side_(side) { }
4 float Square::getArea() const { return side_*side_; }
```

Listado 2.9: `Triangle.h`

```
1  #include <Figure.h>
2
3  class Triangle : public Figure {
4  private:
5    float base_;
6    float height_;
7
8  public:
9    Triangle(float base_, float height_);
10   float getArea() const;
11 };
```

Listado 2.10: `Triangle.cpp`

```
1  #include "Triangle.h"
2
3  Triangle::Triangle (float base, float height) : base_(base),
       height_(height) { }
4  float Triangle::getArea() const { return (base_*height_)/2; }
```

Listado 2.11: `Circle.h`

```
1  #include <Figure.h>
2
3  class Square : public Figure {
4  private:
5    float side_;
6
7  public:
8    Square(float side_length);
9    float getArea() const;
10 };
```

Listado 2.12: `Circle.cpp`

```
1  #include <cmath>
2  #include "Circle.h"
3
4  Circle::Circle(float radious) : radious_(radious) { }
5  float Circle::getArea() const { return radious_*(M_PI*M_PI); }
```

Para generar la biblioteca estática llamada `figures`, es necesario el uso de la herramienta GNU ar:

```
$ g++ -Iheaders -c Square.cpp
$ g++ -Iheaders -c Triangle.cpp
$ g++ -Iheaders -c Circle.cpp
$ ar rs libfigures.a Square.o Triangle.o Circle.o
```

`ar` es un programa que permite, entre otra mucha funcionalidad, empaquetar los archivos de código objeto y generar un índice para crear un biblioteca. Este índice se incluye en el mismo archivo generado y mejora el proceso de enlazado y carga de la biblioteca.

A continuación, se muestra el programa principal que hace uso de la biblioteca:

Listado 2.13: `main.cpp`

```
#include <iostream>

#include <Square.h>
#include <Triangle.h>
#include <Circle.h>

using namespace std;

int main () {
  Square square(5);
  Triangle triangle(5,10);
  Circle circle(10);
  cout << "Square area: " << square.getArea() << endl;
  cout << "Triangle area: " << triangle.getArea() << endl;
  cout << "Circle area: " << circle.getArea() << endl;
  return 0;
}
```

La generación del ejecutable se realizaría de la siguiente manera:

```
$ g++  main.cpp -L. -lfigures -Iheaders -o main
```

Las opciones de enlazado se especifican con `-L` y `-l`. La primera permite añadir un directorio donde se buscarán las bibliotecas. La segunda especifica el nombre de la biblioteca con la que debe enlazarse.

La ruta por defecto en la que se busca las bibliotecas instaladas en el sistema depende de la distribución GNU/Linux que se utilice. Normalmente, se encuentran en `/lib` y `/usr/lib`.

Compilación de una biblioteca dinámica

La generación de una biblioteca dinámica con GCC es muy similar a la de una estática. Sin embargo, el código objeto debe generarse de forma que pueda ser cargado en tiempo de ejecución. Para ello, se debe utilizar la opción `-fPIC` durante la generación de los archivos `.o`. Utilizando el mismo código fuente de la biblioteca `figures`, la compilación se realizaría como sigue:

```
$ g++ -Iheaders -fPIC -c Square.cpp
$ g++ -Iheaders -fPIC -c Triangle.cpp
$ g++ -Iheaders -fPIC -c Circle.cpp
$ g++ -o libfigures.so -shared Square.o Triangle.o Circle.o
```

Como se puede ver, se utiliza GCC directamente para generar la biblioteca dinámica. La compilación y enlazado con el programa principal se realiza de la misma forma que en el caso del enlazado estático. Sin embargo, la ejecución del programa principal es diferente. Al tratarse de código objeto que se cargará en tiempo de ejecución, existen una serie de rutas predefinadas donde se buscarán las bibliotecas. Por defecto, son las mismas que para el proceso de enlazado.

También es posible añadir rutas modificando la variable `LD_LIBRARY_PATH`:

```
$ LD_LIBRARY_PATH=. ./main
```

2.2.5. Otras herramientas

La gran mayoría de las herramientas utilizadas hasta el momento forman parte de la distribución GNU Binutils[1] que se proporcionan en la mayoría de los sistemas GNU/Linux. Existen otras herramientas que se ofrecen en este misma distribución y que pueden ser de utilidad a lo largo del proceso de desarrollo:

- **`c++filt`**: el proceso de *mangling* es el que se realiza cuando se traduce el nombre de las funciones y métodos a bajo nivel. Este mecanismo es útil para realizar la sobreescritura de métodos en C++. `c++filt` es un programa que realiza la operación inversa *demangling*. Es útil para depurar problemas en el proceso de enlazado.
- **`objdump`**: proporciona información avanzada sobre los archivos objetos: símbolos definidos, tamaños, bibliotecas enlazadas dinámicamente, etc. Proporciona una visión detallada y bien organizada por secciones.
- **`readelf`**: similar a `objdump` pero específico para los archivos objeto para plataformas compatibles con el formato binario Executable and Linkable Format (ELF).
- **`nm`**: herramienta para obtener los símbolos definidos y utilizados en los archivos objetos. Muy útil ya que permite listar símbolos definidos en diferentes partes y de distinto tipo (sección de datos, sección de código, símbolos de depuración, etc.)
- **`ldd`**: utilidad que permite mostrar las dependencias de un binario con bibliotecas externas.

2.2.6. Depurando con GDB

Los programas tienen fallos y los programadores cometen errores. Los compiladores ayudan a la hora de detectar errores léxicos, sintácticos y semánticos del lenguaje de entrada. Sin embargo, el compilador no puede deducir (por lo menos hasta hoy) la *lógica* que encierra el programa, su significado final o su propósito. Estos errores se conocen como **errores lógicos**.

Los depuradores son programas que facilitan la labor de detección de errores, sobre todo los lógicos. Con un depurador, el programador puede probar una ejecución paso por paso, examinar/modificar el contenido de las variables en un cierto momento, etc. En general, se pueden realizar las tareas necesarias para conseguir reproducir y localizar un error difícil de detectar a simple vista. Muchos entornos de desarrollo como Eclipse, .NET o Java Beans incorporan un depurador para los lenguajes soportados. Sin duda alguna, se trata de una herramienta esencial en cualquier proceso de desarrollo software.

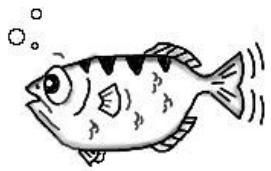


Figura 2.5: GDB es uno de los depuradores más utilizados.

En esta sección se muestra el uso básico de GNU Debugger (GDB) , un depurador libre para sistemas GNU/Linux que soporta diferentes lenguajes de programación, entre ellos C++.

[1] Más información en `http://www.gnu.org/software/binutils/`

Compilar para depurar

GDB necesita información extra que, por defecto, GCC no proporciona para poder realizar las tareas de depuración. Para ello, el código fuente debe ser compilado con la opción `-ggdb`. Todo el código objeto debe ser compilado con esta opción de compilación, por ejemplo:

```
$ g++ -Iheaders -ggdb -c module.cpp
$ g++ -Iheaders -ggdb module.o main.cpp -o main
```

Para depurar **no** se debe hacer uso de las optimizaciones. Éstas pueden generar código que nada tenga que ver con el original.

Arrancando una sesión GDB

Como ejemplo, se verá el siguiente fragmento de código:

Listado 2.14: `main.cpp`

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Test {
6    int _value;
7  public:
8    void setValue(int a) { _value = a; }
9    int getValue() { return _value; }
10 };
11
12 float functionB(string str1, Test* t) {
13   cout << "Function B: " << str1 << ", " << t->getValue() << endl;
14   return 3.14;
15 }
16
17 int functionA(int a) {
18   cout << "Function A: " << a << endl;
19   Test* test = NULL; /** ouch! **/
20   test->setValue(15);
21   cout << "Return B: " << functionB("Hi", test) << endl;
22   return 5;
23 }
24
25 int main() {
26   cout << "Main start" << endl;
27   cout << "Return A: " << functionA(24) << endl;
28   return 0;
29 }
```

La orden para generar el binario con símbolos de depuración sería:

```
$ g++ -ggdb main.cpp -o main
```

Si se ejecuta el código se obtienes la siguiente salida:

```
$ ./main
Main start
Function A: 24
Segmentation fault
```

Una violación de segmento (*segmentation fault*) es uno de los errores lógicos típicos de los lenguajes como C++. El problema es que se está accediendo a una zona de la memoria que no ha sido reservada para el programa, por lo que el sistema operativo interviene denegando ese acceso indebido.

A continuación, se muestra cómo iniciar una sesión de depuración con GDB para encontrar el origen del problema:

```
$ gdb main
...
Reading symbols from ./main done.
(gdb)
```

Como se puede ver, GDB ha cargado el programa, junto con los símbolos de depuración necesarios, y ahora se ha abierto una línea de órdenes donde el usuario puede especificar sus acciones.

Con los programas que fallan en tiempo de ejecución se puede generar un archivo de *core*, es decir, un fichero que contiene un volcado de la memoria en el momento en que ocurrió el fallo. Este archivo puede ser cargado en una sesión de GDB para ser examinado usando la opción `-c`.

Examinando el contenido

Abreviatura

Todas las órdenes de GDB pueden escribirse utilizando su abreviatura. Ej: `run` = `r`.

Para comenzar la ejecución del programa se puede utilizar la orden `start`:

```
(gdb) start
Temporary breakpoint 1 at 0x400d31: file main.cpp, line 26.
Starting program: main

Temporary breakpoint 1, main () at main.cpp:26
26      cout << "Main start" << endl;
(gdb)
```

De esta forma, se ha comenzado la ejecución del programa y se ha detenido en la primera instrucción de la función `main()`. Para reiniciar la ejecución basta con volver a ejecutar `start`.

Para ver más en detalle sobre el código fuente se puede utilizar la orden `list` o símplemente `l`:

```
(gdb) list
21    cout << "Return B: " << functionB("Hi", test) << endl;
22    return 5;
23  }
24
25  int main() {
26    cout << "Main start" << endl;
27    cout << "Return A: " << functionA(24) << endl;
28    return 0;
29  }
(gdb)
```

Como el resto de órdenes, `list` acepta parámetros que permiten ajustar su comportamiento.

Las órdenes que permiten realizar una ejecución controlada son las siguientes:

- `step` (`s`): ejecuta la instrucción actual y salta a la inmediatamente siguiente sin mantener el nivel de la ejecución (*stack frame*), es decir, entra en la definición de la función (si la hubiere).

  `stepi` se comporta igual que `step` pero a nivel de instrucciones máquina.

- `next` (`n`): ejecuta la instrucción actual y salta a la siguiente manteniendo el *stack frame*, es decir, la definición de la función se toma como una instrucción atómica.

  `nexti` se comporta igual que `next` pero si la instrucción es una llamada a función se espera a que termine.

A continuación se va a utilizar `step` para avanzar en la ejecución del programa. Nótese que para repetir la ejecución de `step` basta con introducir una orden vacía. En este caso, GDB vuelve a ejecutar la última orden.

```
(gdb) s
Main start
27    cout << "Return A: " << functionA(24) << endl;
(gdb)
functionA (a=24) at main.cpp:18
18    cout << "Function A: " << a << endl;
(gdb)
```

En este punto se puede hacer uso de las órdenes para mostrar el contenido del parámetro `a` de la función `functionA()`:

```
(gdb) print a
$1 = 24
(gdb) print &a
$2 = (int *) 0x7fffffffe1bc
```

Con `print` y el modificador `&` se puede obtener el contenido y la dirección de memoria de la variable, respectivamente. Con `display` se puede configurar GDB para que muestre su contenido en cada paso de ejecución.

También es posible cambiar el valor de la variable `a`:

```
gdb) set variable a=8080
gdb) print a
3 = 8080
gdb) step
unction A: 8080
9     Test* test = NULL; /** ouch! **/
gdb)
```

La ejecución está detenida en la línea 19 donde un comentario nos avisa del error. Se está creando un puntero con el valor `NULL`. Posteriormente, se invoca un método sobre un objeto que no está convenientemente inicializado, lo que provoca la violación de segmento:

```
(gdb) next
20    test->setValue(15);
(gdb)
```

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400df2 in Test::setValue (this=0x0, a=15) at main.cpp:8
8     void setValue(int a) { _value = a; }
```

Para arreglar este fallo el basta con construir convenientemente el objeto:

Listado 2.15: `functionA` arreglada

```
1 int functionA(int a) {
2   cout << "Function A: " << a << endl;
3   Test* test = new Test();
4   test->setValue(15);
5   cout << "Return B: " << functionB("Hi", test) << endl;
6   return 5;
7 }
```

Breakpoints

La ejecución paso a paso es una herramienta útil para una depuración de grano fino. Sin embargo, si el programa realiza grandes iteraciones en bucles o es demasiado grande, puede ser un poco incómodo (o inviable). Si se tiene la sospecha sobre el lugar donde está el problema se pueden utilizar puntos de ruptura o *breakpoints* que permite detener el flujo del programa en un punto determinado por el usuario.

Con el ejemplo ya arreglado, se configura un breakpoint en la función `functionB()` y otro en la línea 28 con la orden `break`. A continuación, se ejecuta el programa hasta que se alcance el breakpoint con la orden `run` (`r`):

```
(gdb) break functionB
Breakpoint 1 at 0x400c15: file main.cpp, line 13.
(gdb) break main.cpp:28
Breakpoint 2 at 0x400ddb: file main.cpp, line 28.
(gdb) run
Starting program: main
Main start
Function A: 24

Breakpoint 1, functionB (str1=..., t=0x602010) at gdb-fix.cpp:13
13    cout << "Function B: " << str1 << ", " << t->getValue() << endl;
(gdb)
```

¡No hace falta escribir todo!. Utiliza `TAB` para completar los argumentos de una orden.

Con la orden `continue` (`c`) la ejecución avanza hasta el siguiente punto de ruptura (o fin del programa):

```
(gdb) continue
Continuing.
Function B: Hi, 15
Return B: 3.14
Return A: 5

Breakpoint 2, main () at gdb-fix.cpp:28
28    return 0;
```

```
(gdb)
```

Los breakpoint pueden habilitarse, inhabilitarse y/o eliminarse en tiempo de ejecución. Además, GDB ofrece un par de estructuras similares útiles para otras situaciones:

- **Watchpoints**: la ejecución se detiene cuando una determinada expresión cambia.
- **Catchpoints**: la ejecución se detiene cuando se produce un evento, como una excepción o la carga de una librería dinámica.

Stack y frames

En muchas ocasiones, los errores vienen debidos a que las llamadas a funciones no se realizan con los parámetros adecuados. Es común pasar punteros no inicializados o valores incorrectos a una función/método y, por tanto, obtener un error lógico.

Para gestionar las llamadas a funciones y procedimientos, en C/C++ se utiliza la pila (*stack*). En la pila se almacenan *frames*, estructuras de datos que registran las variables creadas dentro de una función así como otra información de contexto. GDB permite manipular la pila y los frames de forma que sea posible identificar un uso indebido de las funciones.

Con la ejecución parada en `functionB()`, se puede mostrar el contenido de la pila con la orden `backtrace` (`bt`):

```
(gdb) backtrace
#0  functionB (str1=..., t=0x602010) at main.cpp:13
#1  0x0000000000400d07 in functionA (a=24) at main.cpp:21
#2  0x0000000000400db3 in main () at main.cpp:27
(gdb)
```

Con `up` y `down` se puede navegar por los frames de la pila, y con `frame` se puede seleccionar uno en concreto:

```
(gdb) up
#1  0x0000000000400d07 in functionA (a=24) at gdb-fix.cpp:21
21    cout << "Return B: " << functionB("Hi", test) << endl;
(gdb)
#2  0x0000000000400db3 in main () at gdb-fix.cpp:27
27    cout << "Return A: " << functionA(24) << endl;
(gdb) frame 0
#0  functionB (str1=..., t=0x602010) at gdb-fix.cpp:13
13    cout << "Function B: " << str1 << ", " << t->getValue() << endl;
(gdb)
```

Una vez seleccionado un frame, se puede obtener toda la información del mismo, además de modificar las variables y argumentos:

Invocar funciones

La orden `call` se puede utilizar para invocar funciones y métodos .

```
(gdb) print *t
$1 = {_value = 15}
(gdb) call t->setValue(1000)
(gdb) print *t
$2 = {_value = 1000}
(gdb)
```

Entornos gráficos para GDB

El aprendizaje de GDB no es sencillo. La interfaz de línea de órdenes es muy potente pero puede ser difícil de asimilar, sobre todo en los primeros pasos del aprendizaje de la herramienta. Por ello, existen diferentes versiones gráficas que, en definitiva, hacen más accesible el uso de GDB:

- **GDB TUI**: normalmente, la distribución de GDB incorpora una interfaz basada en modo texto accesible pulsando Ctrl+x y, a continuación, a.
- **ddd** y **xxgdb**: las librerías gráficas utilizadas son algo anticuadas, pero facilitan el uso de GDB.
- **gdb-mode**: modo de Emacs para GDB. Dentro del modo se puede activar la opción `M-x many-windows` para obtener buffers con toda la información disponible.
- **kdbg**: más atractivo gráficamente (para escritorios KDE).

2.2.7. Construcción automática con GNU Make

En los ejemplos propuestos en la sección 2.2.2 se puede apreciar que el proceso de compilación no es trivial y que necesita de varios pasos para llevarse a cabo. A medida que el proyecto crece es deseable que el proceso de construcción de la aplicación sea lo más automático y fiable posible. Esto evitará muchos errores de compilación a lo largo del proceso de desarrollo.

GNU Make es una herramienta para la construcción de archivos, especificando las **dependencias** con sus archivos fuente. Make es una herramienta genérica, por lo que puede ser utilizada para generar cualquier tipo de archivo desde sus dependencias. Por ejemplo, generar una imagen PNG a partir de una imagen vectorial SVG, obtener la gráfica en JPG de una hoja de cálculo de LibreOffice, etc.

Sin duda, el uso más extendido es la automatización del proceso de construcción de las aplicaciones. Además, Make ofrece la característica de que sólo reconstruye los archivos que han sido modificados, por lo que no es necesario recompilar todo el proyecto cada vez que se realiza algún cambio.

Estructura

Los archivos de Make se suelen almacenar en archivos llamados `Makefile`. La estructura de estos archivos puede verse en el siguiente listado de código:

Listado 2.16: Estructura típica de un archivo Makefile

```
# Variable definitions
VAR1='/home/user'
export VAR2='yes'

# Rules
target1: dependency1 dependency2 ...
    action1
    action2
dependency1: dependency3
    action3
    action4
```

GNU Make tomará como entrada automáticamente el archivo cuyo nombre sea GNUmakefile, makefile o Makefile, en ese orden de prioridad. Puede modificarse el fichero de entrada con `-f`.

Normalmente, el principio del Makefile se reserva para **definiciones de variables** que van a ser utilizadas a lo largo del mismo o por otros Makefiles (para ello puede utilizar `export`). A continuación, se definen el conjunto de **reglas** de construcción para cada uno de los archivos que se pretende generar a partir de sus dependencias. Por ejemplo, el archivo `target1` necesita que existan `dependency1`, `dependency2`, etc. `action1` y `action2` indican cómo se construye. La siguiente regla tiene como objetivo `dependency1` e igualmente se especifica cómo obtenerlo a partir de `dependency3`.

Las acciones de una regla van **siempre** van precedidas de un **tabulado**.

Existen algunos objetivos especiales como `all`, `install` y `clean` que sirven como regla de partida inicial, para instalar el software construido y para limpiar del proyecto los archivos generados, respectivamente.

Tomando como ejemplo la aplicación que hace uso de la biblioteca dinámica, el siguiente listado muestra el Makefile que generaría tanto el programa ejecutable como la biblioteca estática:

Listado 2.17: Makefile básico

```
1
2  all: main
3
4  main:  main.o libfigures.a
5      g++ main.o -L. -lfigures -o main
6
7  main.o: main.cpp
8      g++ -Iheaders -c main.cpp
9
10 libfigures.a: Square.o Triangle.o Circle.o
11     ar rs libfigures.a Square.o Triangle.o Circle.o
12
13 Square.o: Square.cpp
14     g++ -Iheaders -fPIC -c Square.cpp
15
16 Triangle.o: Triangle.cpp
17     g++ -Iheaders -fPIC -c Triangle.cpp
18
19 Circle.o: Circle.cpp
20     g++ -Iheaders -fPIC -c Circle.cpp
21
22 clean:
23     rm -f *.o *.a main
```

Con ello, se puede construir el proyecto utilizando `make [objetivo]`. Si no se proporciona `objetivo` se toma el primero en el archivo. De esta forma, se puede construir todo el proyecto, un archivo en concreto o limpiarlo completamente. Por ejemplo, para construir y limpiar todo el proyecto se ejecutarían las siguientes órdenes:

```
$ make
$ make clean
```

Como ejercicio, se plantean las siguientes preguntas: ¿qué opción permite ejecutar `make` sobre otro archivo que no se llame `Makefile`? ¿Se puede ejecutar `make` sobre un directorio que no sea el directorio actual? ¿Cómo?.

GNU Coding Standars

En el proyecto GNU se definen los objetivos que se esperan en un software que siga estas directrices.

Variables automáticas y reglas con patrones

Make se caracteriza por ofrecer gran versatilidad en su lenguaje. Las variables automáticas contienen valores que dependen del contexto de la regla donde se aplican y permiten definir **reglas genéricas**. Por su parte, los patrones permiten generalizar las reglas utilizando el nombre los archivos generados y los fuentes.

A continuación se presenta una versión mejorada del anterior Makefile haciendo uso de variables, variables automáticas y patrones:

Listado 2.18: Makefile con variables automáticas y patrones

```
1
2  LIB_OBJECTS=Square.o Triangle.o Circle.o
3
4  all: main
5
6  main: main.o libfigures.a
7      g++ $< -L. -lfigures -o $@
8
9  libfigures.a: $(LIB_OBJECTS)
10     ar rs $@ $^
11
12 %.o: %.cpp
13     g++ -Iheaders -c $<
14
15 clean:
16     $(RM) *.o *.a main
```

A continuación se explica cada elemento con maś detalle:

- Variable de usuario `LIB_OBJECTS`: se trata de la lista de archivos objetos que forman la biblioteca. Al contenido se puede acceder utilizando el operador `$()`.
- Variable predefinada `RM`: con el valor `rm -f`. Se utiliza en el objetivo `clean`.
- Variables automáticas `$@`, `$<`, `$^`: se utiliza para hacer referencia al **nombre del objetivo** de la regla, al nombre de la **primera dependencia** y a la lista de **todas las dependencias** de la regla, respectivamente.
- Regla con patrón: en línea 12 se define una regla genérica a través de un patrón en el que se define cómo generar cualquier archivo objeto `.o`, a partir de un archivo fuente `.cpp`.

Como ejercicio se plantean las siguientes cuestiones: ¿qué ocurre si una vez construido el proyecto se modifica algún fichero `.cpp`? ¿Y si se modifica una cabecera `.h`? ¿Se podría construir una regla con patrón genérica para construir la biblioteca estática? ¿Cómo lo harías?.

Reglas implícitas

Como se ha dicho, GNU Make es ampliamente utilizado para la construcción de proyectos software donde están implicados diferentes procesos de compilación y generación de código. Por ello, proporciona las llamadas *reglas implícitas* de forma que si se definen ciertas variables de usuario, Make puede deducir cómo construir los objetivos.

A continuación, se transforma el ejemplo anterior utilizando las reglas implícitas:

Listado 2.19: Makefile con reglas implícitas

```
1  CC=g++
2  CXXFLAGS=-Iheaders
3  LDFLAGS=-L.
4  LDLIBS=-lfigures
5
6  LIB_OBJECTS=Square.o Triangle.o Circle.o
7
8  all: libfigures.a main
9
10 libfigures.a: $(LIB_OBJECTS)
11     $(AR) r $@ $^
12
13 clean:
14     $(RM) *.o *.a main
```

Como se puede ver, Make puede generar automáticamente los archivos objeto `.o` a partir de la coincidencia con el nombre del fichero fuente (que es lo habitual). Por ello, no es necesario especificar cómo construir los archivos `.o` de la biblioteca, ni siquiera la regla para generar `main` ya que asume de que se trata del ejecutable (al existir un fichero llamado `main.cpp`).

Las variables de usuario que se han definido permiten configurar los flags de compilación que se van a utilizar en las reglas explícitas. Así:

- `CC`: la orden que se utilizará como compilador. En este caso, el de C++.
- `CXXFLAGS`: los flags para el preprocesador y compilador de C++ (ver sección 2.2.3). Aquí sólo se define la opción `-I`, pero también es posible añadir optimizaciones y la opción de depuración `-ggdb` o `-fPIC` para las bibliotecas dinámicas.
- `LDFLAGS`: flags para el enlazador (ver sección 2.2.3). Aquí se definen las rutas de búsqueda de las bibliotecas estáticas y dinámicas.
- `LDLIBS`: en esta variable se especifican las opciones de enlazado. Normalmente, basta con utilizar la opción `-l` con las bibliotecas necesarias para generar el ejecutable.

En GNU/Linux, el programa `pkg-config` permite conocer los flags de compilación y enlazado de una biblioteca determinada.

Funciones

GNU Make proporciona un conjunto de funciones que pueden ser de gran ayuda a la hora de construir los Makefiles. Muchas de las funciones están diseñadas para el tratamiento de cadenas, ya que se suelen utilizar para transformar los nombres de archivos. Sin embargo, existen muchas otras como para realizar ejecución condicional, bucles y ejecutar órdenes de consola. En general, las funciones tienen el siguiente formato:

```
$(nombre arg1,arg2,arg3,...)
```

Las funciones se pueden utilizar en cualquier punto del Makefile, desde las acciones de una regla hasta en la definición de un variable. En el siguiente listado se muestra el uso de algunas de estas funciones:

Listado 2.20: Makefile con reglas implícitas

```
1  CC=g++
2
3  ifeq ($(DEBUG),yes)
4      CXXFLAGS=-Iheader -Wall -ggdb
5  else
6      CXXFLAGS=-Iheader -O2
7  endif
8
9  LDFLAGS=-L.
10 LDLIBS=-lfigures
11
12 LIB_OBJECTS=Square.o Triangle.o Circle.o
13
14 all: libfigures.a main
15     $(info All done!)
16
17 libfigures.a: $(LIB_OBJECTS)
18     $(AR) r $@ $^
19     $(warning Compiled objects from $(foreach OBJ,
20                 $(LIB_OBJECTS),
21                 $(patsubst  %.o, %.cpp, $(OBJ))))
22
23 clean:
24     $(RM) *.o *.a main
25     $(shell find  -name '*~' -delete)
```

Las funciones que se han utilizado se definen a continuación:

- **Funciones condicionales**: funciones como `ifeq` o `ifneq` permiten realizar una ejecución condicional. En este caso, si existe una variable de entorno llamada `DEBUG` con el valor `yes`, los flags de compilación se configuran en consecuencia.

  Para definir la variable `DEBUG`, es necesario ejecutar `make` como sigue:

  ```
  $ DEBUG=yes make
  ```

- **Funciones de bucles**: `foreach` permite aplicar una función a cada valor de una lista. Este tipo de funciones devuelven una lista con el resultado.

- **Funciones de tratamiento de texto**: por ejemplo, `patsubst` toma como primer parámetro un patrón que se comprobará por cada `OBJ`. Si hay *matching*, será sustituido por el patrón definido como segundo parámetro. En definitiva, cambia la extensión `.o` por `.cpp`.

- **Funciones de log**: `info`, `warning`, `error`, etc. permiten mostrar texto a diferente nivel de severidad.
- **Funciones de consola**: `shell` es la función que permite ejecutar órdenes en un terminal. La salida es útil utilizarla como entrada a una variable.

Más información

GNU Make es una herramienta que está en continuo crecimiento y esta sección sólo ha sido una pequeña presentación de sus posibilidades. Para obtener más información sobre las funciones disponibles, otras variables automáticas y objetivos predefinidos se recomiendo utilizar el manual en línea de Make[2], el cual siempre se encuentra actualizado.

2.3. Gestión de proyectos y documentación

Los proyectos software pueden ser realizados por varios equipos de personas, con formación y conocimientos diferentes, que deben *colaborar* entre sí. Además, una componente importante del proyecto es la *documentación* que se genera durante el transcurso del mismo, es decir, cualquier documento escrito o gráfico que permita entender mejor los componentes del mismo y, por ello, asegure una mayor mantenibilidad para el futuro (manuales, diagramas, especificaciones, etc.).

La *gestión del proyecto* es un proceso transversal al resto de procesos y tareas y se ocupa de la planificación y asignación de los recursos de los que se disponen. Se trata de un proceso *dinámico*, ya que debe adaptarse a los diferentes cambios e imprevistos que puedan surgir durante el desarrollo. Para detectar estos cambios a tiempo, dentro de la gestión del proyecto se realizan *tareas de seguimiento* que consisten en registrar y notificar errores y retrasos en las diferentes fases y entregas.

Existen muchas herramientas que permiten crear **entornos colaborativos** que facilitan el trabajo tanto a desarrolladores como a los jefes de proyecto, los cuales están más centrados en las tareas de gestión. En esta sección se presentan algunos entornos colaborativos actuales, así como soluciones específicas para un proceso concreto.

2.3.1. Sistemas de control de versiones

El resultado más importante de un proyecto software son los archivos de distinto tipo que se generan; desde el código fuente, hasta los diseños, bocetos y documentación del mismo. Desde el punto de vista técnico, se trata de gestionar una cantidad importante de archivos que son modificados a lo largo del tiempo por diferentes personas. Además, es posible que para un conjunto de archivos sea necesario volver a una versión anterior.

Por ejemplo, un fallo de diseño puede tener como consecuencia que se genere un código que no es escalable y difícil de mantener. Si es posible volver a un estado original conocido (por ejemplo, antes de tomarse la decisión de diseño), el tiempo invertido en *revertir los cambios* es menor.

[2] http://www.gnu.org/software/make/manual/make.html

Por otro lado, sería interesante tener la posibilidad de realizar *desarrollos en paralelo* de forma que exista una versión «estable» de todo el proyecto y otra más «experimental» del mismo donde se probaran diferentes algoritmos y diseños. De esta forma, probar el impacto que tendría nuevas implementaciones sobre el proyecto no afectaría a una versión más «oficial». También es común que se desee añadir una nueva funcionalidad y ésta se realiza en paralelo junto con otros desarrollos.

Los **sistemas de control de versiones** o Version Control System (VCS) permiten gestionar los archivos de un proyecto (y sus versiones) y que sus integrantes puedan acceder remotamente a ellos para descargarlos, modificarlos y publicar los cambios. También se encargan de detectar posibles conflictos cuando varios usuarios modifican los mismos archivos y de proporcionar un sistema básico de registro de cambios.

Como norma general, al VCS debe subirse el archivo fuente y nunca el archivo generado. No se deben subir binarios ya que no es fácil seguir la pista a sus modificaciones.

Sistemas centralizados vs. distribuidos

Existen diferentes criterios para clasificar los diferentes VCS existentes. Uno de los que más influye tanto en la organización y uso del repositorio es si se trata de VCS centralizado o distribuido. En la figura 2.6 se muestra un esquema de ambas filosofías.

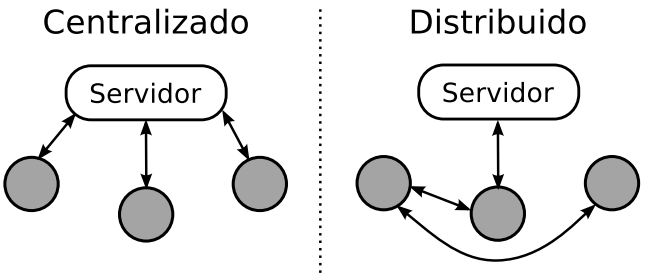


Figura 2.6: Esquema centralizado vs. distribuido de VCS.

Los VCS centralizados como CVS o *Subversion* se basan en que existe un nodo servidor con el que todos los clientes conectan para obtener los archivos, subir modificaciones, etc. La principal ventaja de este esquema reside en su sencillez: las diferentes versiones del proyecto están únicamente en el servidor central, por lo que los posibles conflictos entre las modificaciones de los clientes pueden detectarse y gestionarse más fácilmente. Sin embargo, el servidor es un único punto de fallo y en caso de caída, los clientes quedan aislados.

Por su parte, en los VCS distribuidos como *Mercurial* o *Git*, cada cliente tiene un repositorio local al nodo en el que se suben los diferentes cambios. Los cambios pueden agruparse en *changesets*, lo que permite una gestión más ordenada. Los clientes actúan de servidores para el resto de los componentes del sistema, es decir, un cliente puede descargarse una versión concreta de otro cliente.

Esta arquitectura es tolerante a fallos y permite a los clientes realizar cambios sin necesidad de conexión. Posteriormente, pueden sincronizarse con el resto. Aún así, un VCS distribuido puede utilizarse como uno centralizado si se fija un nodo como servidor, pero se perderían algunas posibilidades que este esquema ofrece.

Subversion

Figura 2.7: Logotipo del proyecto Apache Subversion.

Subversion (SVN) es uno de los VCS centralizado más utilizado, probablemente, debido a su sencillez en el uso. Básicamente, los clientes tienen accesible en un servidor todo el repositorio y es ahí donde se envían los cambios.

Para crear un repositorio, en el servidor se puede ejecutar la siguiente orden:

```
$ svnadmin create /var/repo/myproject
```

Esto creará un árbol de directorios en `/var/repo/myproject` que contiene toda la información necesaria. Una vez hecho esto es necesario hacer accesible este directorio a los clientes. En lo que sigue, se supone que los usuarios tienen acceso a la máquina a través de una cuenta SSH, aunque se pueden utilizar otros métodos de acceso.

Es recomendable que el acceso al repositorio esté controlado. HTTPS o SSH son buenas opciones de métodos de acceso.

Inicialmente, los clientes pueden descargarse el repositorio por primera vez utilizando la orden `checkout`:

```
$ svn checkout svn+ssh://user1@myserver:/var/repo/myproject
Checked out revision X.
```

Esto ha creado un directorio `myproject` con el contenido del repositorio. Una vez descargado, se pueden realizar todos los cambios que se deseen y, posteriormente, subirlos al repositorio. Por ejemplo, añadir archivos y/o directorios:

```
$ mkdir doc
$ echo "This is a new file" > doc/new_file
$ echo "Other file" > other_file
$ svn add doc other_file
A         doc
A         doc/new_file
A         other_file
```

La operación `add` indica qué archivos y directorios han sido seleccionados para ser añadidos al repositorio (marcados con `A`). Esta operación **no** sube efectivamente los archivos al servidor. Para subir cualquier cambio se debe hacer un *commit*:

```
$ svn commit
```

A coninuación, se lanza un editor de texto[3] para que se especifique un mensaje que describa los cambios realizados. Además, también incluye un resumen de todas las operaciones que se van a llevar a cabo en el commit (en este caso, sólo se añaden elementos). Una vez terminada la edición, se salva y se sale del editor y la carga comenzará.

Cada commit aumenta en 1 el número de revisión. Ese número será el que podremos utilizar para volver a versiones anteriores del proyecto utilizando:

[3] El configurado en la variable de entorno EDITOR.

```
$ svn update -r REVISION
```

Si no se especifica la opción `-r`, la operación `update` trae la última revisión (*head*).

En caso de que otro usuario haya modificado los mismos archivos y lo haya subido antes al repositorio central, al hacerse el commit se detectará un *conflicto*. Como ejemplo, supóngase que el cliente `user2` ejecuta lo siguiente:

```
$ svn checkout svn+ssh://user2@myserver:/var/repo/myproject
$ echo "I change this file" > other_file
$ svn commit
Committed revision X+1.
```

Y que el cliente `user1`, que está en la versión `X`, ejecuta lo siguiente:

```
$ echo "I change the content" > doc/new_file
$ svn remove other_file
D         other_file
$ svn commit
svn: Commit failed (details follow):
svn: File 'other_file' is out of date
```

Para resolver el conflicto, el cliente `user1` debe actualizar su versión:

```
$ svn update
C other_file
At revision X+1.
```

La marca `C` indica que `other_file` queda en conflicto y que debe resolverse manualmente. Para resolverlo, se debe editar el archivo donde Subversion marca las diferencias con los símbolos `'<'` y `'>'`.

También es posible tomar como solución el revertir los cambios realizados por `user1` y, de este modo, aceptar los de `user2`:

```
$ svn revert other_file
$ svn commit
Committed revision X+2.
```

Nótese que este commit sólo añade los cambios hechos en `new_file`, aceptando los cambios en `other_file` que hizo `user2`.

Figura 2.8: Logotipo del proyecto Mercurial.

Mercurial

Como se ha visto, en los VCS centralizados como Subversion no se permite, por ejemplo, que los clientes hagan commits si no están conectados con el servidor central. Los VCS como Mercurial (HG), permiten que los clientes tengan un repositorio local, con su versión modificada del proyecto y la sincronización del mismo con otros servidores (que pueden ser también clientes).

Para crear un repositorio Mercurial, se debe ejecutar lo siguiente:

```
$ hg init /home/user1/myproyect
```

Al igual que ocurre con Subversion, este directorio debe ser accesible mediante algún mecanismo (preferiblemente, que sea seguro) para que el resto de usuarios pueda acceder. Sin embargo, el usuario `user1` puede trabajar directamente sobre ese directorio.

Para obtener una versión inicial, otro usuario (`user2`) debe *clonar* el repositorio. Basta con ejecutar lo siguiente:

```
$ hg clone ssh://user2@host//home/user1/myproyect
```

A partir de este instante, `user2` tiene una versión inicial del proyecto extraída a partir de la del usuario `user1`. De forma muy similar a Subversion, con la orden `add` se pueden añadir archivos y directorios.

Mientras que en el modelo de Subversion, los clientes hacen `commit` y `update` para subir cambios y obtener la última versión, respectivamente; en Mercurial es algo más complejo, ya que existe un *repositorio local*. Como se muestra en la figura 2.9, la operación `commit` (3) sube los cambios a un repositorio local que cada cliente tiene.

Cada commit se considera un *changeset*, es decir, un conjunto de cambios agrupados por un mismo ID de revisión. Como en el caso de Subversion, en cada commit se pedirá una breve descripción de lo que se ha modificado.

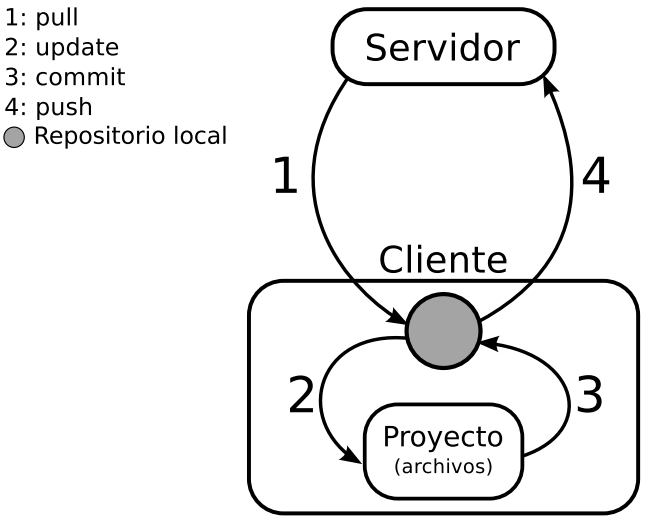


Figura 2.9: Esquema del flujo de trabajo básico en Mercurial

Una vez hecho todos commits, para llevar estos cambios a un servidor remoto se debe ejecutar la orden de `push` (4). Siguiendo con el ejemplo, el cliente `user2` lo enviará por defecto al repositorio del que hizo la operación `clone`.

El sentido inverso, es decir, traerse los cambios del servidor remoto a la copia local, se realiza también en 2 pasos: `pull` (1) que trae los cambios del repositorio remoto al repositorio local; y `update` (2), que aplica dichos cambios del repositorio local al directorio de trabajo. Para hacer los dos pasos al mismo tiempo, se puede hacer lo siguiente:

```
$ hg pull -u
```

Para evitar conflictos con otros usuarios, una buena costumbre **antes** de realizar un `push` es conveniente obtener los posibles cambios en el servidor con `pull` y `update`.

Para ver cómo se gestionan los conflictos en Mercurial, supóngase que `user1` realiza lo siguiente:

```
$ echo "A file" > a_file
$ hg add a_file
$ hg commit
```

Al mismo tiempo, `user2` ejecuta lo siguiente:

```
$ echo "This is one file" > a_file
$ hg add a_file
$ hg commit
$ hg push
abort: push creates new remote head xxxxxx!
(you should pull and merge or use push -f to force)
```

Al intentar realizar el `push` y entrar en conflicto, Mercurial avisa de ello deteniendo la carga. En este punto se puede utilizar la opción `-f` para forzar la operación de `push`, lo cual crearía un nuevo *head*. Como resultado, se crearía una nueva rama a partir de ese conflicto de forma que se podría seguir desarrollando omitiendo el conflicto. Si en el futuro se pretende unir los dos heads se utilizan las operaciones `merge` y `resolve`.

La otra solución, y normalmente la más común, es obtener los cambios con `pull`, unificar heads (`merge`), resolver los posibles conflictos manualmente si es necesario (`resolve`), hacer commit de la solución dada (`commit`) y volver a intentar la subida (`push`):

`hgview`

`hgview` es una herramienta gráfica que permite visualizar la evolución de las ramas y heads de un proyecto que usa Mercurial.

```
$ hg pull
adding changesets
adding manifests
adding file changes
added 2 changesets with 1 changes to 1 files (+1 heads)
(run 'hg heads' to see heads, 'hg merge' to merge)
$ hg merge
merging a_file
warning: conflicts during merge.
merging a_file failed!
0 files updated, 0 files merged, 0 files removed, 1 files unresolved
$ hg resolve -a
$ hg commit
$ hg push
```

Para realizar cómodamente la tarea de resolver los conflictos manualmente existen herramientas como `meld` que son invocadas automáticamente por Mercurial cuando se encuentran conflictos de este tipo.

Git

Diseñado y desarrollado por Linus Torvalds para el proyecto del kernel Linux, Git es un VCS distribuido que cada vez es más utilizado por la comunidad de desarrolladores. En términos generales, tiene una estructura similar a Mercurial: independencia entre repositorio remotos y locales, gestión local de cambios, etc.

Sin embargo, Git es en ocasiones preferido sobre Mercurial por algunas de sus características propias:

- Eficiencia y rapidez a la hora de gestionar grandes cantidades de archivos. Git no funciona peor conforme la historia del repositorio crece.
- Facilita el desarrollo *no lineal*, es decir, el programador puede crear ramas locales e integrar los cambios entre diferentes ramas (tanto remotas como locales) de una forma simple y con muy poco coste. Git está diseñado para que los cambios puedan ir de rama a rama y ser revisados por diferentes usuarios.
- Diseñado como un conjunto de pequeñas herramientas, la mayoría escritas en C, que pueden ser compuestas entre ellas para hacer tareas más complejas.

En general, Git proporciona más flexibilidad al usuario, permitiendo hacer tareas complejas y de grano fino (como la división de un cambio en diferentes cambios) y al mismo tiempo es eficiente y escalable para proyectos con gran cantidad de archivos y usuarios concurrentes.

Figura 2.10: Logotipo del proyecto Git.

Un repositorio Git está formado, básicamente, por un conjunto de objetos *commit* y un conjunto de referencias a esos objetos llamadas *heads*. Un *commit* es un concepto similar a un *changeset* de Mercurial y se compone de las siguientes partes:

- El conjunto de ficheros que representan al proyecto en un momento concreto.
- Referencias a los *commits* padres.
- Un nombre formado a partir del contenido (usando el algoritmo SHA1).

Cada *head* apunta a un *commit* y tiene un nombre simbólico para poder ser referenciado. Por defecto, todos los respositorios Git tienen un *head* llamado *master*. *HEAD* (nótese todas las letras en mayúscula) es una referencia al *head* usado en cada instante. Por lo tanto, en un repositorio Git, en un estado sin cambios, *HEAD* apuntará al *master* del respositorio.

Para crear un repositorio donde sea posible que otros usuarios puedan subir cambios se utiliza la orden `init`:

```
$ git init --bare /home/user1/myproyect
```

La opción `–bare` indica a Git que se trata de un repositorio que no va a almacenar una copia de trabajo de usuario, sino que va a actuar como sumidero de cambios de los usuarios del proyecto. Este directorio deberá estar accesible para el resto de los usuarios utilizando algún protocolo de comunicación soportado por Git (ssh, HTTP, etc.).

El resto de usuarios pueden obtener una copia del repositorio utilizando la siguiente orden:

```
$ git clone ssh://user2@host/home/user1/myproyect
```

A modo de ejemplo ilustrativo, se podría realizar la operación `clone` de la siguiente forma:

```
$ git init /home/user2/myproyect
$ cd /home/user2/myproyect
$ git remote add -t master origin ssh://user2@host/home/user1/
    myproyect
$ git pull
```

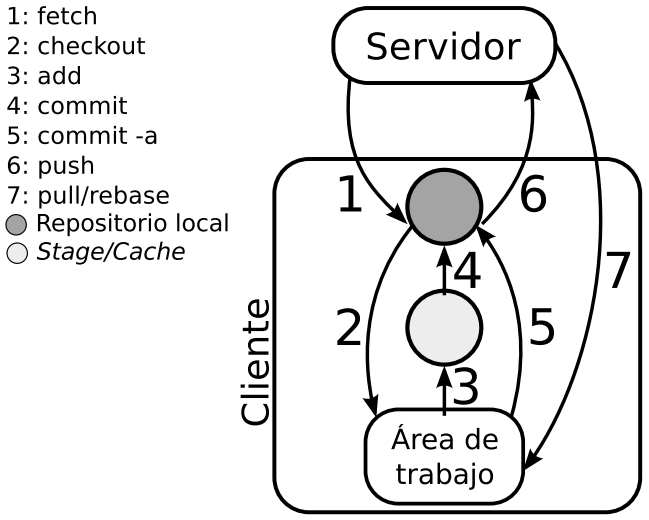


Figura 2.11: Esquema del flujo de trabajo básico en Git

De esta manera, una vez creado un repositorio Git, es posible reconfigurar en la URL donde se conectarán las órdenes `pull` y `fetch` para descargar el contenido. En la figura 2.11 se muestra un esquema general del flujo de trabajo y las órdenes asociadas.

En Git se introduce el espacio *stage* (también llamado *index* o *cache*) que actúa como paso intermedio entre la copia de trabajo del usuario y el repositorio local. Sólo se pueden enviar cambios (*commits*) al repositorio local si éstos están previamente en el *stage*. Por ello, todos los nuevos archivos y/o modificaciones que se realicen deben ser «añadidas» al *stage* antes.

La siguiente secuencia de comandos añaden un archivo nuevo al repositorio local. Nótese que `add` sólo añade el archivo al *stage*. Hasta que no se realice `commit` no llegará a estar en el repositorio local:

```
$ echo "Test example" > example
$ git add example
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   example
#

$ git commit -m "Test example: initial version"
[master 2f81676] Test example: initial version
1 file changed, 1 insertion(+)
create mode 100644 example

$ git status
# On branch master
nothing to commit, working directory clean
```

Nótese como la última llamada a `status` no muestra ningún cambio por subir al repositorio local. Esto significa que la copia de trabajo del usuario está sincronizada con el repositorio local (todavía no se han realizado operaciones con el remoto). Se puede utilizar `reflog` para ver la historia:

```
$ git reflog
2f81676 HEAD@{0}: commit: Test example: initial version
...
```

`diff` se utiliza para ver cambios entre *commits*, ramas, etc. Por ejemplo, la siguiente orden muestra las diferencias entre *master* del repositorio local y *master* del remoto:

```
$ git diff master origin/master
diff --git a/example b/example
new file mode 100644
index 0000000..ac19bf2
--- /dev/null
+++ b/example
@@ -0,0 +1 @@
+Test example
```

Para modificar código y subirlo al repositorio local se sigue el mismo procedimiento: (1) realizar la modificación, (2) usar `add` para añadir el archivo cambiado, (3) hacer `commit` para subir el cambio al repositorio local. Sin embargo, como se ha dicho anteriormente, una buena característica de Git es la creación y gestión de ramas (*branches*) locales que permiten hacer un desarrollo en paralelo. Esto es muy útil ya que, normalmente, en el ciclo de vida de desarrollo de un programa se debe simultanear tanto la creación de nuevas características como arreglos de errores cometidos. En Git, estas ramas no son más que referencias a *commits*, por lo que son muy ligeras y pueden llevarse de un sitio a otro de forma sencilla.

`gitk`

Para entender mejor estos conceptos y visualizarlos durante el proceso de desarrollo, existen herramientas gráficas como `gitk` que permiten ver todos los *commits* y *heads* en cada instante.

Como ejemplo, la siguiente secuencia de órdenes crea una rama local a partir del *HEAD*, modifica un archivo en esa rama y finalmente realiza un *merge* con *master*:

```
$ git checkout -b "NEW-BRANCH"
Switched to a new branch 'NEW-BRANCH'

$ git branch
* NEW-BRANCH
master

$ emacs example  # se realizan las modificaciones
$ git add example
$ git commit -m "remove 'example'"
[NEW-BRANCH 263e915] remove 'example'
1 file changed, 1 insertion(+), 1 deletion(-)

$ git checkout master
Switched to branch 'master'
$ git branch
NEW-BRANCH
* master

$ git merge NEW-BRANCH
Updating 2f81676..263e915
Fast-forward
example | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

Nótese cómo la orden `branch` muestra la rama actual marcada con el símbolo `*`. Utilizando las órdenes `log` y `show` se pueden listar los *commits* recientes. Estas órdenes aceptan, además de identificadores de *commits*, ramas y rangos temporales de forma que pueden obtenerse gran cantidad de información de ellos.

Finalmente, para desplegar los cambios en el repositorio remoto sólo hay que utilizar:

```
$ git push
```

Git utiliza ficheros como `.gitconfig` y `.gitignore` para cargar configuraciones personalizadas e ignorar ficheros a la hora de hacer los *commits*, respectivamente. Son muy útiles. Revisa la documentación de `git-config` y `gitignore` para más información.

2.3.2. Documentación

Uno de los elementos más importantes que se generan en un proyecto es la documentación: cualquier elemento que permita entender mejor tanto el proyecto en su totalidad como sus partes, de forma que facilite el proceso de **mantenimiento** en el futuro. Además, una buena documentación hará más sencilla la **reutilización** de componentes.

Existen muchos formatos de documentación que pueden servir para un proyecto software. Sin embargo, muchos de ellos, tales como PDF, ODT, DOC, etc., son formatos «binarios» por lo que no son aconsejables para utilizarlos en un VCS. Además, utilizando texto plano es más sencillo crear programas que automaticen la generación de documentación, de forma que se ahorre tiempo en este proceso.

Por ello, aquí se describen algunas formas de crear documentación basadas en texto plano. Obviamente, existen muchas otras y, seguramente, sean tan válidas como las que se proponen aquí.

Doxygen

Doxygen es un sistema que permite generar la documentación utilizando analizadores de código que averiguan la estructura de módulos y clases, así como las funciones y los métodos utilizados. Además, se pueden realizar anotaciones en los comentarios del código que sirven para añadir información más detallada. La principal ventaja es que se vale del propio código fuente para generar la documentación. Además, si se añaden comentarios en un formato determinado, es posible ampliar la documentación generada con notas y aclaraciones sobre las estructuras y funciones utilizadas.

Algunos piensan que el uso de programas como Doxygen es bueno porque «obliga» a comentar el código. Otros piensan que no es así ya que los comentarios deben seguir un determinado formato, dejando de ser comentarios propiamente dichos.

El siguiente fragmento de código muestra una clase en C++ documentada con el formato de Doxygen:

Listado 2.21: Clase con comentarios Doxygen

```
1 /**
2    This is a test class to show Doxygen format documentation.
3   */
4
5 class Test {
6 public:
7   /// The Test constructor.
```

```
8    /**
9       \param s the name of the Test.
10   */
11   Test(string s);
12
13   /// Start running the test.
14   /**
15      \param max maximum time of test delay.
16      \param silent if true, do not provide output.
17      \sa Test()
18   */
19   int run(int max, bool silent);
20 };
```

Por defecto, Doxygen genera la documentación en HTML y basta con ejecutar la siguiente orden en el raíz del código fuente para obtener una primera aproximación:

```
$ doxygen .
```

reStructuredText

reStructuredText (RST) es un formato de texto básico que permite escribir texto plano añadiendo pequeñas anotaciones de formato de forma que no se pierda legibilidad. Existen muchos traductores de RST a otros formatos como PDF (`rst2pdf`) y HTML (`rst2html`), que además permiten modificar el estilo de los documentos generados.

El formato RST es similar a la sintaxis de los sistema tipo wiki. Un ejemplo de archivo en RST puede ser el siguiente:

Listado 2.22: Archivo en RST

```
1  ================
2   The main title
3  ================
4
5  This is an example of document in ReStructured Text (RST). You can
       get
6  more info about RST format at `RST Reference
7  <http://
       docutils.sourceforge.net/docs/ref/rst/restructuredtext.html>`_.
8
9  Other section
10 =============
11
12 You can use bullet items:
13
14  - Item A
15
16  - Item B
17
18 And a enumerated list:
19
20  1. Number 1
21
22  2. Number 2
23
24 Tables
25 ------
26
27 +--------------+----------+-----------+-----------+
28 | row 1, col 1 | column 2 | column 3  | column 4  |
29 +--------------+----------+-----------+-----------+
```

```
| row 2        |          |           |           |
+--------------+----------+-----------+-----------+

Images
------

.. image:: gnu.png
   :scale: 80
   :alt: A title text
```

Como se puede ver, aunque RST añade una sintaxis especial, el texto es completamente legible. Ésta es una de las ventajas de RST, el uso de etiquetas de formato que no «ensucian» demasiado el texto.

YAML

YAML (YAML Ain't Markup Language)[4] es un lenguaje diseñado para serializar datos procedentes de aplicaciones en un formato que sea legible para los humanos. Estrictamente, no se trata de un sistema para documentación, sin embargo, y debido a lo cómodo de su sintaxis, puede ser útil para exportar datos, cargar los mismos en el programa, representar configuraciones, etc. Otra de sus ventajas es que hay un gran número de bibliotecas en diferentes lenguajes (C++, Python, Java, etc.) para tratar información YAML. Las librerías permiten automáticamente salvar las estructuras de datos en formato YAML y el proceso inverso: cargar estructuras de datos a partir del YAML.

En el ejemplo siguiente, extraído de la documentación oficial, se muestra una factura. De un primer vistazo, se puede ver qué campos forman parte del tipo de dato factura tales como `invoice`, `date`, etc. Cada campo puede ser de distintos tipo como numérico, booleano o cadena de caracteres, pero también listas (como `product`) o referencias a otros objetos ya declarados (como `ship-to`).

Listado 2.23: Archivo en YAML

```
--- !<tag:clarkevans.com,2002:invoice>
invoice: 34843
date   : 2001-01-23
bill-to: &id001
    given  : Chris
    family : Dumars
    address:
        lines: |
            458 Walkman Dr.
            Suite #292
        city    : Royal Oak
        state   : MI
        postal  : 48046
ship-to: *id001
product:
    - sku         : BL394D
      quantity    : 4
      description : Basketball
      price       : 450.00
    - sku         : BL4438H
      quantity    : 1
      description : Super Hoop
      price       : 2392.00
tax  : 251.42
total: 4443.52
comments:
```

[4] http://www.yaml.org

```
27      Late afternoon is best.
28      Backup contact is Nancy
29      Billsmer @ 338-4338.
```

2.3.3. Forjas de desarrollo

Hasta ahora, se han mostrado herramientas específicas que permiten crear y gestionar los elementos más importantes de un proyecto software: los archivos que lo forman y su documentación. Sin embargo, existen otras como la gestión de tareas, los mecanismos de notificación de errores o los sistemas de comunicación con los usuarios que son de utilidad en un proyecto.

Las forjas de desarrollo son sistemas colaborativos que integran no sólo herramientas básicas para la gestión de proyectos, como un VCS, sino que suelen proporcionar herramientas para:

- **Planificación y gestión de tareas**: permite anotar qué tareas quedan por hacer y los plazos de entrega. También suelen permitir asignar prioridades.
- **Planificación y gestión de recursos**: ayuda a controlar el grado de ocupación del personal de desarrollo (y otros recursos).
- **Seguimiento de fallos**: también conocido como *bug tracker*, es esencial para llevar un control sobre los errores encontrados en el programa. Normalmente, permiten gestionar el ciclo de vida de un fallo, desde que se descubre hasta que se da por solucionado.
- **Foros**: normalmente, las forjas de desarrollo permiten administrar varios foros de comunicación donde con la comunidad de usuarios del programa pueden escribir propuestas y notificar errores.

Las forjas de desarrollo suelen ser accesibles via web, de forma que sólo sea necesario un navegador para poder utilizar los diferentes servicios que ofrece. Dependiendo de la forja de desarrollo, se ofrecerán más o menos servicios. Sin embargo, los expuestos hasta ahora son los que se proporcionan habitualmente. Existen forjas gratuitas en Internet que pueden ser utilizadas para la creación de un proyecto. Algunas de ellas:

- **GNA**[5]: es una forja de desarrollo creada por la Free Software Foundation de Francia que soporta, actualmente, repositorios CVS, GNU Arch y Subversion. Los nuevos proyectos son estudiados cuidadosamente antes de ser autorizados.
- **Launchpad**[6]: forja gratuita para proyectos de software libre creada por Canonical Ltd. Se caracteriza por tener un potente sistema de *bug tracking* y proporcionar herramientas automáticas para despliegue en sistemas Debian/Ubuntu.
- **BitBucket**[7]: forja de la empresa Atlassian que ofrece repositorios Mercurial y Git. Permite crear proyectos privados gratuitos pero con límite en el número de desarrolladores por proyecto.
- **GitHub**[8]: forja proporcionada por GitHub Inc. que utiliza repositorios Git. Es gratuito siempre que el proyecto sea público, es decir, pueda ser descargado y modificado por cualquier usuario de Github sin restricción.

[5] http://gna.org
[6] https://launchpad.net/
[7] http://bitbucket.org
[8] http://github.com

- **SourceForge**[9]: probablemente, una de las forjas gratuitas más conocidas. Propiedad de la empresa GeekNet Inc., soporta Subversion, Git, Mercurial, Bazaar y CVS.
- **Google Code**[10]: la forja de desarrollo de Google que soporta Git, Mercurial y Subversion.

Redmine

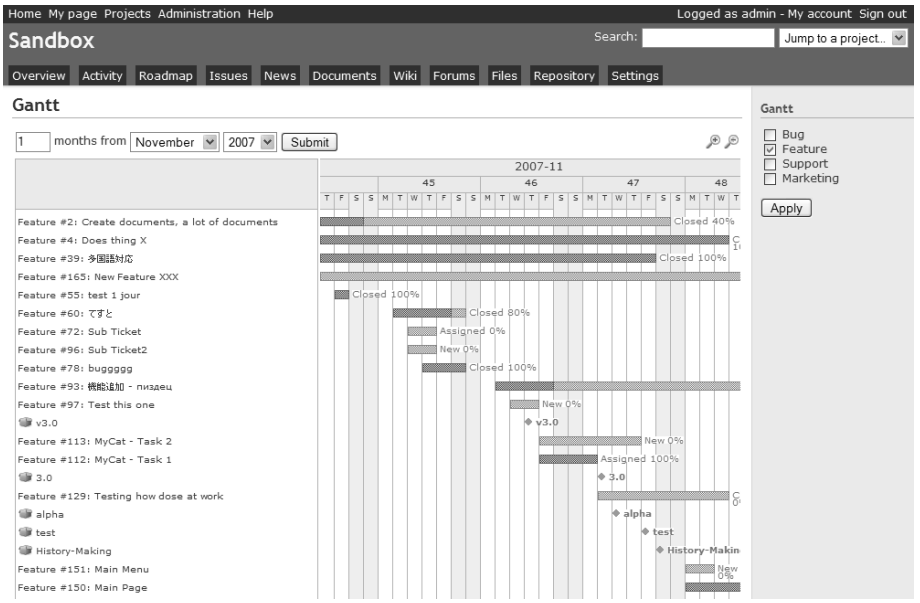


Figura 2.12: Aspecto de la herramienta de gestión de tareas de Redmine

Además de los servicios gratuitos presentados, existe gran variedad de software que puede ser utilizado para gestionar un proyecto de forma que pueda ser instalado en un servidor personal y así no depender de un servicio externo. Tal es el caso de Redmine (véase figura 2.12) que entre las herramientas que proporciona cabe destacar las siguientes características:

- Permite crear **varios proyectos**. También es configurable qué servicios se proporcionan en cada proyecto: gestor de tareas, tracking de fallos, sistema de documentación wiki, etc.
- **Integración con repositorios**, es decir, el código es accesible a través de Redmine y se pueden gestionar tareas y errores utilizando los comentarios de los commits.
- **Gestión de usuarios** que pueden utilizar el sistema y sus políticas de acceso.
- Está construido en **Ruby** y existe una amplia variedad de **plugins** que añaden funcionalidad extra.

[9] `http://sourceforge.net`
[10] `http://code.google.com`

Capítulo 3

C++. Aspectos Esenciales

David Vallejo Fernández

El lenguaje más utilizado para el desarrollo de videojuegos comerciales es C++, debido especialmente a su potencia, eficiencia y portabilidad. En este capítulo se hace un recorrido por C++ desde los aspectos más básicos hasta las herramientas que soportan la POO (Programación Orientada a Objetos) y que permiten diseñar y desarrollar código que sea reutilizable y mantenible, como por ejemplo las plantillas y las excepciones.

3.1. Utilidades básicas

POO

La programación orientada a objetos tiene como objetivo la organización eficaz de programas. Básicamente, cada componente es un objeto autocontenido que tiene una serie de operaciones y de datos o estado. Este planteamiento permite reducir la complejidad y gestionar grandes proyectos de programación.

En esta sección se realiza un recorrido por los aspectos básicos de C++, haciendo especial hincapié en aquellos elementos que lo diferencian de otros lenguajes de programación y que, en ocasiones, pueden resultar más complicados de dominar por aquellos programadores inexpertos en C++.

3.1.1. Introducción a C++

C++ se puede considerar como el lenguaje de programación más importante en la actualidad. De hecho, algunas autores relevantes [81] consideran que si un programador tuviera que aprender un único lenguaje, éste debería ser C++. Aspectos como su sintaxis y su filosofía de diseño definen elementos clave de programación, como por ejemplo la orientación a objetos. C++ no sólo es importante por sus propias características, sino también porque ha sentado las bases para el desarrollo de futuros lenguajes de programación. Por ejemplo, Java o C# son descendientes directos de C++. Desde el punto de vista profesional, C++ es sumamente importante para cualquier programador.

El **origen de C++** está ligado al origen de C, ya que C++ está construido sobre C. De hecho, C++ es un superconjunto de C y se puede entender como una versión extendida y mejorada del mismo que integra la filosofía de la POO y otras mejoras, como por ejemplo la inclusión de un conjunto amplio de bibliotecas. Algunos autores consideran que C++ surgió debido a la necesidad de tratar con programas de mayor complejidad, siendo impulsado en gran parte por la POO.

C++ fue diseñado por **Bjarne Stroustrup**[1] en 1979. La idea de Stroustrup fue añadir nuevos aspectos y mejoras a C, especialmente en relación a la POO, de manera que un programador de C sólo que tuviera que aprender aquellos aspectos relativos a la OO.

En el caso particular de la industria del videojuego, C++ se puede considerar como el estándar *de facto* debido principalmente a su **eficiencia y portabilidad**. C++ es una de los pocos lenguajes que posibilitan la programación de alto nivel y, de manera simultánea, el acceso a los recursos de bajo nivel de la plataforma subyacente. Por lo tanto, C++ es una mezcla perfecta para la programación de sistemas y para el desarrollo de videojuegos. Una de las principales claves a la hora de manejarlo eficientemente en la industria del videojuego consiste en encontrar el equilibrio adecuado entre eficiencia, fiabilidad y mantenibilidad [27].

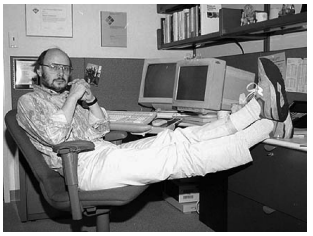


Figura 3.1: Bjarne Stroustrup, creador del lenguaje de programación C++ y personaje relevante en el ámbito de la programación.

Dominar C++

Una de las mayores ventajas de C++ es que es extremadamente potente. Sin embargo, utilizarlo eficientemente es difícil y su curva de aprendizaje no es gradual.

3.1.2. ¡Hola Mundo! en C++

A continuación se muestra el clásico *¡Hola Mundo!* implementado en C++. En este primer ejemplo, se pueden apreciar ciertas diferencias con respecto a un programa escrito en C.

Listado 3.1: Hola Mundo en C++

```
1  /* Mi primer programa con C++. */
2
3  #include <iostream>
4  using namespace std;
5
6  int main () {
7
8    string nombre;
9
10   cout << "Por favor, introduzca su nombre... ";
11   cin >> nombre;
12   cout << "Hola " << nombre << "!"<< endl;
13
14   return 0;
15
16 }
```

La directiva *include* de la línea 3 incluye la biblioteca <*iostream*>, la cual soporta el sistema de E/S de C++. A continuación, en la 4 el programa le indica al compilador que utilice el espacio de nombres *std*, en el que se declara la biblioteca estándar de C++. Un **espacio de nombres** delimita una zona de declaración en la que incluir diferentes elementos de un programa. Los espacios de nombres siguen la misma filosofía que los *paquetes* en Java y tienen como objetivo organizar los programas. El hecho de utilizar un espacio de nombres permite acceder a sus elementos y funcionalidades sin tener que especificar a qué espacio pertenecen.

[1] http://www2.research.att.com/~bs/

En la línea 10 de hace uso de *cout* (*console output*), la sentencia de salida por consola junto con el operador $<<$, redireccionando lo que queda a su derecha, es decir, *Por favor, introduzca su nombre...*, hacia la salida por consola. A continuación, en la siguiente línea se hace uso de *cin* (*console input*) junto con el operador $>>$ para redirigir la entrada proporcionado por teclado a la variable *nombre*. Note que dicha variable es de tipo cadena, un tipo de datos de C++ que se define como un array de caracteres finalizado con el carácter *null*. Finalmente, en la línea 12 se saluda al lector, utilizando además la sentencia *endl* para añadir un retorno de carro a la salida y limpiar el *buffer*.

3.1.3. Tipos, declaraciones y modificadores

En C++, los tipos de datos básicos y sus declaraciones siguen un esquema muy similar al de otros lenguajes de programación, como por ejemplo Java. En este caso, existen siete tipos de datos básicos: *carácter*, *carácter amplio*, *entero*, *punto flotante*, *punto flotante de doble precisión*, *lógico o booleano* y *nulo*. Recuerde que en C++ es necesario declarar una variable antes de utilizarla para que el compilador conozca el tipo de datos asociado a la variable. Al igual que ocurre en otros lenguajes de programación, las variables pueden tener un alcance global o local (por ejemplo en una función).

En el caso particular de los tipos nulos o *void*, cabe destacar que éstos se utilizan para las funciones con tipo de retorno nulo o como tipo base para punteros a objetos de tipo desconocido a priori.

C++ permite modificar los tipos *char*, *int* y *double* con los modificadores *signed*, *unsigned long*, y *short*, con el objetivo de incrementar la precisión en función de las necesidades de la aplicación. Por ejemplo, el tipo de datos *double* se puede usar con el modificador *long*, y no así con el resto.

C++. Tipado.

C++ es un lenguaje de programación con tipado estático, es decir, la comprobación de tipos se realiza en tipo de compilación y no en tiempo de ejecución.

Por otra parte, C++ también posibilita el concepto de constante mediante la palabra reservada ***const***, con el objetivo de expresar que un valor no cambia de manera directa. Por ejemplo, muchos objetos no cambian sus valores después de inicializarlos. Además, las constantes conducen a un código más mantenible en comparación con los valores literales incluidos directamente en el código. Por otra parte, muchos punteros también se suelen utilizar solamente para operaciones de lectura y, particularmente, los parámetros de una función no se suelen modificar sino que simplemente se consultan.

Como regla general, se debería delegar en el compilador todo lo posible en relación a la detección de errores.

El uso de *const* también es ventajoso respecto a la directiva de preprocesado *#define*, ya que permite que el compilador aplique la típica prevención de errores de tipo. Esto posibilita detectar errores en tiempo de compilación y evitar problemas potenciales. Además, la depuración es mucho más sencilla por el hecho de manejar los nombres simbólicos de las constantes.

Desreferencia

El acceso al objeto al que señala un puntero es una de las operaciones fundamentales y se denomina indirección o desreferencia, indistintamente.

3.1.4. Punteros, arrays y estructuras

Un **puntero** es una variable que almacena una dirección de memoria, típicamente la localización o dirección de otra variable. Si *p* contiene la dirección de *q*, entonces *p apunta* a *q*. Los punteros se declaran igual que el resto de variables y están asociados a un tipo específico, el cual ha de ser válido en C++. Un ejemplo de declaración de puntero a entero es

```
int *ip;
```

En el caso de los punteros, el operador unario * es el utilizado para desreferenciarlos y acceder a su contenido, mientras que el operador unario **&** se utiliza para acceder a la dirección de memoria de su operando. Por ejemplo,

```
edadptr = &edad;
```

asigna la dirección de memoria de la variable *edad* a la variable *edadptr*. Recuerde que con el operador * es posible acceder al contenido de la variable a la que apunta un puntero. Por ejemplo,

```
miEdad = *edadptr
```

permite almacenar el contenido de *edad* en *miEdad*.

Indirección múltiple

Un puntero a puntero es un caso de indirección múltiple. El primer puntero contiene la dirección de memoria de otro puntero, mientras que éste contiene la dirección de memoria de una variable.

Cuando el compilador de C++ encuentra una cadena literal, la almacena en la tabla de cadenas del programa y genera un puntero a dicha cadena.

En C++ existe una relación muy estrecha entre los **punteros y los arrays**, siendo posible intercambiarlos en la mayoría de casos. El siguiente listado de código muestra un sencillo ejemplo de indexación de un array mediante aritmética de punteros.

Listado 3.2: Indexación de un array

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5
6    char s[20] = "hola mundo";
7    char *p;
8    int i;
9
10   for (p = s, i = 0; p[i]; i++)
11     p[i] = toupper(p[i]);
12
13   cout << s << endl;
14
15   return 0;
16
17 }
```

En la inicialización del bucle *for* de la línea 10 se aprecia cómo se asigna la dirección de inicio del array *s* al puntero *p* para, posteriormente, indexar el array mediante *p* para resaltar en mayúsculas el contenido del array una vez finalizada la ejecución del bucle. Note que C++ permite la inicialización múltiple.

Los punteros son enormemente útiles y potentes. Sin embargo, cuando un puntero almacena, de manera accidental, valores incorrectos, el proceso de depuración puede resultar un auténtico quebradero de cabeza. Esto se debe a la propia naturaleza del puntero y al hecho de que indirectamente afecte a otros elementos de un programa, lo cual puede complicar la localización de errores.

El caso típico tiene lugar cuando un puntero apunta a algún lugar de memoria que no debe, modificando datos a los que no debería apuntar, de manera que el programa muestra resultados indeseables posteriormente a su ejecución inicial. En estos casos, cuando se detecta el problema, encontrar la evidencia del fallo no es una tarea trivial, ya que inicialmente puede que no exista evidencia del puntero que provocó dicho error. A continuación se muestran algunos de los errores típicos a la hora de manejar punteros [81].

1. No inicializar punteros. En el listado que se muestra a continuación, *p* contiene una dirección desconocida debido a que nunca fue definida. En otras palabras, no es posible conocer dónde se ha escrito el valor contenido en *edad*.

Listado 3.3: Primer error típico. No inicializar punteros.

```
1 int main () {
2   int edad, *p;
3
4   edad = 23;
5   *p = edad; // p?
6
7   return 0;
8 }
```

En un programa que tenga una mayor complejidad, la probabilidad de que *p* apunte a otra parte de dicho programa se incrementa, con la más que probable consecuencia de alcanzar un resultado desastroso.

Punteros y funciones

Recuerde que es posible utilizar punteros a funciones, es decir, se puede recuperar la dirección de memoria de una función para, posteriormente, llamarla. Este tipo de punteros se utilizan para manejar rutinas que se puedan aplicar a distintos tipos de objetos, es decir, para manejar el *polimorfismo*.

2. Comparar punteros de forma no válida. La comparación de punteros es, generalmente, inválida y puede causar errores. En otras palabras, no se deben realizar suposiciones sobre qué dirección de memoria se utilizará para almacenar los datos, si siempre será dicha dirección o si distintos compiladores tratarán estos aspectos del mismo modo. No obstante, si dos punteros apuntan a miembros del mismo arreglo, entonces es posible compararlos. El siguiente fragmento de código muestra un ejemplo de uso incorrecto de punteros en relación a este tipo de errores.

Listado 3.4: Segundo error típico. Comparación incorrecta de punteros.

```
1  int main () {
2    int s[10];
3    int t[10];
4    int *p, *q;
5
6    p = s;
7    q = t;
8
9    if (p < q) {
10     // ...
11   }
12
13   return 0;
14 }
```

No olvide que una de las claves para garantizar un uso *seguro* de los punteros consiste en conocer en todo momento hacia dónde están apuntando.

3. Olvidar el *reseteo* de punteros. El siguiente listado de código muestra otro error típico, que se puede resumir en no controlar el comportamiento de un puntero. Básicamente, la primera vez que se ejecuta el bucle, *p* apunta al comienzo de *s*. Sin embargo, en la segunda iteración *p* continua incrementándose debido a que su valor no se ha establecido al principio de *s*. La solución pasa por mover la línea 11 al bucle *do-while*.

Listado 3.5: Tercer error típico. Olvidar el *reseteo* de un puntero.

```
1  #include <iostream>
2  #include <cstdio>
3  #include <cstring>
4  using namespace std;
5
6  int main () {
7
8    char s[100];
9    char *p;
10
11   p = s;
12
13   do {
14     cout << "Introduzca una cadena... ";
15     fgets(p, 100, stdin);
16
17     while (*p)
18       cout << *p++ << " ";
19
20     cout << endl;
21   }while (strcmp(s, "fin"));
22
23   return 0;
24
25 }
```

Finalmente, una **estructura** es un tipo agregado de datos que se puede definir como una colección de variables que mantienen algún tipo de relación lógica. Obviamente, antes de poder crear objetos de una determinada estructura, ésta ha de definirse.

El listado de código siguiente muestra un ejemplo de declaración de estructura y de su posterior manipulación mediante el uso de punteros. Recuerde que el acceso a los campos de una estructura se realiza mediante el operador *flecha* **->** en caso de acceder a ellos mediante un puntero, mientras que el operador punto **.** se utiliza cuando se accede de manera directa.

Nótese el uso del modificador *const* en el segundo parámetro de la función *modificar_nombre* que, en este caso, se utiliza para informar al compilador de que dicha variable no se modificará internamente en dicha función. Asimismo, en dicha función se hace uso del operador **&** en relación a la variable *nuevo_nombre*. En la siguiente subsección se define y explica un nuevo concepto que C++ proporciona para la especificación de parámetros y los valores de retorno: las **referencias**.

Punteros y *const*

El uso de punteros y *const* puede dar lugar a confusión, en función del lugar en el que se sitúe dicha palabra clave. Recuerde que *const* siempre se refiere a lo que se encuentra inmediatamente a la derecha. Así, *int* const p* es un puntero constante, pero no así los datos a los que apunta.

Listado 3.6: Definición y uso de estructuras.

```
1  #include <iostream>
```

```
2  using namespace std;
3
4  struct persona {
5    string nombre;
6    int edad;
7  };
8
9  void modificar_nombre (persona *p, const string& nuevo_nombre);
10
11 int main () {
12   persona p;
13   persona *q;
14
15   p.nombre = "Luis";
16   p.edad = 23;
17   q = &p;
18
19   cout << q->nombre << endl;
20   modificar_nombre(q, "Sergio");
21   cout << q->nombre << endl;
22
23   return 0;
24 }
25
26 void modificar_nombre (persona *p, const string& nuevo_nombre) {
27   p->nombre = nuevo_nombre;
28 }
```

3.1.5. Referencias y funciones

En general, existen dos formas de pasar argumentos a una función. La primera se denomina ***paso por valor*** y consiste en pasar una copia del valor del argumento como parámetro a una función. Por lo tanto, los cambios realizados por la función no afectan a dicho argumento. Por otra parte, el ***paso por referencia*** permite la copia de la dirección del argumento (y no su valor) en el propio parámetro. Esto implica que los cambios efectuados sobre el parámetro afectan al argumento utilizado para realizar la llamada a la función.

Paso de parámetros

Es muy importante distinguir correctamente entre paso de parámetros por valor y por referencia para obtener el comportamiento deseado en un programa y para garantizar que la eficiencia del mismo no se vea penalizada.

Por defecto, C++ utiliza el paso por valor. Sin embargo, el paso por referencia se puede realizar utilizando punteros, pasando la dirección de memoria del argumento externo a la función para que ésta lo modifique (si así está diseñada). Este enfoque implica hacer un uso explícito de los operadores asociados a los punteros, lo cual implica que el programador ha de pasar las direcciones de los argumentos a la hora de llamar a la función. Sin embargo, en C++ también es posible indicar de manera automática al compilador que haga uso del paso por referencia: mediante el uso de **referencias**.

Una referencia es simplemente un nombre alternativo para un objeto. Este concepto tan sumamente simple es en realidad extremadamente útil para gestionar la complejidad. Cuando se utiliza una referencia, la dirección del argumento se pasa automáticamente a la función de manera que, dentro de la función, la referencia se desreferencia automáticamente, sin necesidad de utilizar punteros. Las referencias se declaran precediendo el operador **&** al nombre del parámetro.

Las referencias son muy parecidas a los punteros. Se refieren a un objeto de manera que las operaciones afectan al objeto al cual apunta la referencia. La creación de referencias, al igual que la creación de punteros, es una operación muy eficiente.

El siguiente listado de código muestra la típica función *swap* implementada mediante el uso de referencias. Como se puede apreciar, los parámetros pasados por referencia no hacen uso en ningún momento del operador *, como ocurre con los punteros. En realidad, el compilador genera automáticamente la dirección de los argumentos con los que se llama a *swap*, desreferenciando a ambos.

Listado 3.7: Función *swap* con referencias.

```
1  #include <iostream>
2  using namespace std;
3
4  void swap (int &a, int &b);
5
6  int main () {
7    int x = 7, y = 13;
8
9    cout << "[" << x << ", " << y << "]" << endl; // Imprime [7, 13].
10   swap(x, y);
11   cout << "[" << x << ", " << y << "]" << endl; // Imprime [13, 7].
12
13   return 0;
14 }
15
16 void swap (int &a, int &b) {
17   int aux;
18
19   aux = a; // Guarda el valor al que referencia a.
20   a = b;   // Asigna el valor de b a a.
21   b = aux; // Asigna el valor de aux a b.
22 }
```

En C++, existen ciertas **diferencias relevantes** entre referencias y punteros [27]:

- A la hora de trabajar con una referencia, la sintaxis utilizada es la misma que con los objetos. En otras palabras, en lugar de desreferenciar con el operador flecha **->**, para acceder a variables y funciones se utiliza el operador punto.
- Las referencias sólo se pueden inicializar una vez. Por el contrario, un puntero puede apuntar a un determinado objeto y, posteriormente, apuntar a otro distinto. Sin embargo, una vez inicializada una referencia, ésta no se puede cambiar, comportándose como un puntero constante.
- Las referencias han de inicializarse tan pronto como sean declaradas. Al contrario que ocurre con los punteros, no es posible crear una referencia y esperar para después inicializarla.
- Las referencias no pueden ser nulas, como consecuencia directa de los dos puntos anteriores. Sin embargo, esto no quiere decir que el elemento al que referencian siempre sea válido. Por ejemplo, es posible borrar el objeto al que apunta una referencia e incluso *truncarla* mediante algún molde para que apunte a *null*.
- Las referencias no se pueden crear o eliminar como los punteros. En ese sentido, son iguales que los objetos.

Las **funciones** también pueden devolver referencias. En C++, una de las mayores utilidades de esta posibilidad es la sobrecarga de operadores. Sin embargo, en el listado que se muestra a continuación se refleja otro uso potencial, debido a que cuando se devuelve una referencia, en realidad se está devolviendo un puntero implícito al valor de retorno. Por lo tanto, es posible utilizar la función en la parte izquierda de una asignación.

Funciones y referencias

En una función, las referencias se pueden utilizar como parámetros de entrada o como valores de retorno.

Como se puede apreciar en el siguiente listado, la función *f* devuelve una referencia a un valor en punto flotante de doble precisión, en concreto a la variable global *valor*. La parte importante del código está en la línea 15, en la que *valor* se actualiza a 7,5, debido a que la función devuelve dicha referencia.

Listado 3.8: Retorno de referencias.

```
1  #include <iostream>
2  using namespace std;
3
4  double &f ();
5
6  double valor = 10.0;
7
8  int main () {
9    double nuevo_valor;
10
11   cout << f() << endl;
12   nuevo_valor = f();
13   cout << nuevo_valor << endl;
14
15   f() = 7.5;
16   cout << f() << endl;
17
18   return 0;
19 }
20
21 double &f () { return valor; }
```

Aunque se discutirá más adelante, las referencias también se pueden utilizar para devolver objetos desde una función de una manera eficiente. Sin embargo, hay que ser cuidadoso con la referencia a devolver, ya que si se asigna a un objeto, entonces se creará una copia. El siguiente fragmento de código muestra un ejemplo representativo vinculado al uso de matrices de 16 elementos, estructuras de datos típicamente utilizada en el desarrollo de videojuegos.

Listado 3.9: Retorno de referencias. Copia de objetos

```
1  const Matrix4x4 &GameScene::getCameraRotation () const
2  {
3    return c_rotation; // Eficiente. Devuelve una referencia.
4  }
5
6  // Cuidado! Se genera una copia del objeto.
7  Matrix4x4 rotation = camera.getCameraRotation;
8
9  // Eficiente.
10 Matrix4x4 &rotation = camera.getCameraRotation;
```

Las **ventajas de las referencias** sobre los punteros se pueden resumir en que utilizar referencias es una forma de más alto nivel de manipular objetos, ya que permite al desarrollador *olvidarse* de los detalles de gestión de memoria y centrarse en la lógica del problema a resolver. Aunque pueden darse situaciones en las que los punteros son más adecuados, una buena regla consiste en utilizar referencias siempre que sea posible, ya que su sintaxis es más limpia que la de los punteros y su uso es menos proclive a errores.

Siempre que sea posible, es conveniente utilizar referencias, debido a su sencillez, manejabilidad y a la ocultación de ciertos aspectos como la gestión de memoria.

Quizá el uso más evidente de un puntero en detrimento de una referencia consiste en la creación o eliminación de objetos de manera dinámica. En este caso, el mantenimiento de dicho puntero es responsable de su creador. Si alguien hace uso de dicho puntero, debería indicar explícitamente que no es responsable de su liberación.

Por otra parte, si se necesita cambiar el objeto al que se referencia, entonces también se hace uso de punteros, ya que las referencias no se pueden reasignar. En otros casos, el desarrollador también puede asumir el manejo de un puntero nulo, devolviéndolo cuando una función generó algún error o incluso para manejar parámetros que sean opcionales. En estos casos, las referencias tampoco se pueden utilizar.

Finalmente, otra razón importante para el manejo de punteros en lugar de referencias está vinculada a la aritmética de punteros, la cual se puede utilizar para iterar sobre una región de memoria. Sin embargo, este mecanismo de bajo nivel tiene el riesgo de generar *bugs* y su mantenimiento puede ser tedioso. En general, debería evitarse cuando así sea posible. En la práctica, la aritmética de punteros se puede justificar debido a su elevada eficiencia en lugar de realizar una iteración que garantice la integridad de los tipos [27].

3.2. Clases

3.2.1. Fundamentos básicos

En el ámbito de la POO, las **clases** representan una manera de asociar datos con funcionalidad. Los **objetos** son las instancias específicas de una clase, de manera que cada una tiene sus propios datos pero todas ellas comparten la misma funcionalidad a nivel de clase.

La parte de datos en una clase de C++ no difiere de una estructura en C. Sin embargo, C++ ofrece tres **niveles de acceso** a los datos: públicos, privados o protegidos. Por defecto, los *miembros de una clase* son privados, mientras que en una estructura son públicos.

Destrucción de objetos

Recuerde que los objetos creados dentro de un bloque se destruyen cuando dicho bloque se abandona por el flujo del programa. Por el contrario, los objetos globales se destruyen cuando el programa finaliza su ejecución.

Debido a que la mayoría de los objetos requieren una inicialización de su estado, C++ permite inicializar los objetos cuando estos son creados mediante el uso de **constructores**. Del mismo modo, C++ contempla el concepto de **destructor** para contemplar la posibilidad de que un objeto realice una serie de operaciones antes de ser destruido. El siguiente listado de código muestra la especificación de una clase en C++ con los miembros de dicha clase, su visibilidad, el constructor, el destructor y otras funciones.

Listado 3.10: Clase Figura

```
1  class Figura
2  {
3  public:
4    Figura (double i, double j);
5    ~Figura ();
6
7    void setDim (double i, double j);
8    double getX () const;
9    double getY () const;
10
11 protected:
12   double _x, _y;
13 };
```

Note cómo las variables de clase se definen como protegidas, es decir, con una visibilidad privada fuera de dicha clase a excepción de las clases que hereden de *Figura*, tal y como se discutirá en la sección 3.3.1. El constructor y el destructor comparten el nombre con la clase, pero el destructor tiene delante el símbolo **~**.

Paso por referencia

Recuerde utilizar parámetros por referencia *const* para minimizar el número de copias de los mismos.

El resto de funciones sirven para modificar y acceder al estado de los objetos instanciados a partir de dicha clase. Note el uso del modificador **const** en las funciones de acceso *getX()* y *getY()*, con el objetivo de informar de manera explícita al compilador de que dichas funciones no modifican el estado de los objetos. A continuación, se muestra la implementación de las funciones definidas en la clase *Figura*.

Recuerde estructurar adecuadamente su código y seguir un convenio de nombrado que facilite su mantenibilidad. Si se integra en un proyecto activo, procure seguir el convenio previamente adoptado.

Uso de *inline*

El modificador *inline* se suele incluir después de la declaración de la función para evitar líneas de código demasiado largas (siempre dentro del archivo de cabecera). Sin embargo, algunos compiladores obligan a incluirlo en ambos lugares.

Antes de continuar discutiendo más aspectos de las clases, resulta interesante introducir brevemente el concepto de **funciones en línea** (*inlining*), una técnica que puede reducir la sobrecarga implícita en las llamadas a funciones. Para ello, sólo es necesario incluir el modificador *inline* delante de la declaración de una función. Esta técnica permite obtener exactamente el mismo rendimiento que el acceso directo a una variable sin tener que *desperdiciar* tiempo en ejecutar la llamada a la función, interactuar con la pila del programa y volver de dicha función.

Las funciones en línea no se pueden usar indiscriminadamente, ya que pueden degradar el rendimiento de la aplicación fácilmente. En primer lugar, el tamaño del ejecutable final se puede disparar debido a la duplicidad de código. Así mismo, la caché de código también puede hacer que dicho rendimiento disminuya debido a las continuas penalizaciones asociadas a incluir tantas funciones en línea. Finalmente, los tiempos de compilación se pueden incrementar en grandes proyectos.

Listado 3.11: Clase Figura (implementación)

```
1  #include "Figura.h"
2
3  Figura::Figura
4  (double i, double j)
5  {
6    _x = i;
7    _y = j;
8  }
9
10 Figura::~Figura ()
11 {
12 }
13
14 void
15 Figura::setDim
16 (double i, double j)
17 {
18   _x = i;
19   _y = j;
20 }
21
22 double
23 Figura::getX () const
24 {
25   return _x;
26 }
27
28 double
29 Figura::getY () const
30 {
31   return _y;
32 }
```

Una buena regla para usar de manera adecuada el modificador *inline* consiste en evitar su uso hasta prácticamente completar el desarrollo de un proyecto. A continuación, se puede utilizar alguna herramienta de *profiling* para detectar si alguna función sencilla está entre las más utilidas. Este tipo de funciones son candidatas potenciales para modificarlas con *inline* y, en consecuencia, elementos para mejorar el rendimiento del programa.

Al igual que ocurre con otros tipos de datos, los objetos también se pueden manipular mediante **punteros**. Simplemente se ha de utilizar la misma notación y recordar que la aritmética de punteros también se puede usar con objetos que, por ejemplo, formen parte de un array.

Para los objetos creados en memoria dinámica, el operador **new** invoca al constructor de la clase de manera que dichos objetos existen hasta que explícitamente se eliminen con el operador **delete** sobre los punteros asociados. A continuación se muestra un listado de código que hace uso de la clase *Figura* previamente introducida.

Listado 3.12: Manipulación de objetos con punteros

```
1  #include <iostream>
```

```
2  #include "Figura.h"
3  using namespace std;
4
5  int main () {
6    Figura *f1;
7    f1 = new Figura (1.0, 0.5);
8
9    cout << "[" << f1->getX() << ", " << f1->getY() << "]" << endl;
10
11   delete f1;
12   return 0;
13 }
```

3.2.2. Aspectos específicos de las clases

En esta subsección se discutirán algunos aspectos de C++ relacionados con el concepto de clase que resultan muy útiles en una gran variedad de situaciones y pueden contribuir a mejorar la calidad y la mantenibilidad del código fuente generado por un desarrollador.

En primer lugar, se discutirá el concepto de **función amiga** de una clase. Un función amiga de una o varias clases, especificada así con el modificador *friend* y sin ser una de sus funciones miembro, puede acceder a los miembros privados de la misma. Aunque en principio puede parecer que esta posibilidad no ofrece ninguna ventaja sobre una función miembro, en realidad sí que puede aportar beneficios desde el punto de vista del diseño. Por ejemplo, este tipo de funciones pueden ser útiles para sobrecargar ciertos tipos de operadores y pueden simplificar la creación de algunas funciones de entrada/salida.

Un uso bastante común de las funciones amigas se da cuando existen dos o más clases con miembros que de algún modo están relacionados. Por ejemplo, imagine dos clases distintas que hacen uso de un recurso común cuando se da algún tipo de evento externo. Por otra parte, otro elemento del programa necesita conocer si se ha hecho uso de dicho recurso antes de poder utilizarlo para evitar algún tipo de inconsistencia futura. En este contexto, es posible crear una función en cada una de las dos clases que compruebe, consultado una variable booleana, si dicho recurso fue utilizado, provocando dos llamadas independientes. Si esta situación se da continuamente, entonces se puede llegar a producir una sobrecarga de llamadas.

Por el contrario, el uso de una función amiga permitiría comprobar de manera directa el estado de cada objeto mediante una única llamada que tenga acceso a las dos clases. En este tipo de situaciones, las funciones amigas contribuyen a un **código más limpio y mantenible**. El siguiente listado de código muestra un ejemplo de este tipo de situaciones.

En el ejemplo, línea 21, se puede apreciar cómo la función recibe dos objetos como parámetros. Al contrario de lo que ocurre en otros lenguajes como Java, en C++ los objetos, por defecto, se pasan por valor. Esto implica que en realidad la función recibe una copia del objeto, en lugar del propio objeto con el que se realizó la llamada inicial. Por tanto, los cambios realizados dentro de la función no afectan a los argumentos. Aunque el paso de objetos es un procedimiento sencillo, en realidad se generan ciertos eventos que pueden sorprender inicialmente. El siguiente listado de código muestra un ejemplo.

La salida del programa es la siguiente:

```
Construyendo...
7
Destruyendo...
Destruyendo...
```

Listado 3.13: Ejemplo de uso de funciones amigas

```
1  const int USADO = 1;
2  const int NO_USADO = 0;
3
4  class B;
5
6  class A {
7    int _estado;
8
9  public:
10   A() {_estado = NO_USADO;}
11   void setEstado (int estado) {_estado = estado;}
12   friend int usado (A a, B b);
13 };
14
15 class B {
16   int _estado;
17
18 public:
19   B() {_estado = NO_USADO;}
20   void setEstado (int estado) {_estado = estado;}
21   friend int usado (A a, B b);
22 };
23
24 int usado (A a, B b) {
25   return (a._estado || b._estado);
26 }
```

Como se puede apreciar, existe una llamada al constructor al crear *a* (línea 24) y dos llamadas al destructor. Como se ha comentado antes, cuando un objeto se pasa a una función, entonces se crea una copia del mismo, la cual se destruye cuando finaliza la ejecución de la función. Ante esta situación surgen dos preguntas: i) ¿se realiza una llamada al constructor? y ii) ¿se realiza una llamada al destructor?

En realidad, lo que ocurre cuando se pasa un objeto a una función es que se llama al **constructor de copia**, cuya responsabilidad consiste en definir cómo se copia un objeto. Si una clase no tiene un constructor de copia, entonces C++ proporciona uno por defecto, el cual crea una copia bit a bit del objeto. En realidad, esta decisión es bastante lógica, ya que el uso del constructor normal para copiar un objeto no generaría el mismo resultado que el estado que mantiene el objeto actual (generaría una copia con el estado inicial).

Sin embargo, cuando una función finaliza y se ha de eliminar la copia del objeto, entonces se hace uso del destructor debido a que la copia se encuentra fuera de su ámbito local. Por lo tanto, en el ejemplo anterior se llama al destructor tanto para la copia como para el argumento inicial.

El paso de objetos no siempre es seguro. Por ejemplo, si un objeto utilizado como argumento reserva memoria de manera dinámica, liberándola en el destructor, entonces la copia local dentro de la función liberará la misma región de memoria al llamar a su destructor. Este tipo de situaciones puede causar errores potenciales en un programa. La solución más directa pasa por utilizar un puntero o una referencia en lugar del propio objeto. De este modo, el destructor no se llamará al volver de la función.

Listado 3.14: Paso de objetos por valor

```
1  #include <iostream>
2  using namespace std;
3
4  class A {
5    int _valor;
6  public:
7    A(int valor): _valor(valor) {
8      cout << "Construyendo..." << endl;
9    }
10   ~A() {
11     cout << "Destruyendo..." << endl;
12   }
13
14   int getValor () const {
15     return _valor;
16   }
17 };
18
19 void mostrar (A a) {
20   cout << a.getValor() << endl;
21 }
22
23 int main () {
24   A a(7);
25   mostrar(a);
26   return 0;
27 }
```

En el caso de devolver objetos al finalizar la ejecución de una función se puede producir un problema similar, ya que el objeto temporal que se crea para almacenar el valor de retorno también realiza una llamada a su destructor. La solución pasa por devolver un puntero o una referencia, pero en casos en los que no sea posible el constructor de copia puede contribuir a solventar este tipo de problemas.

No devuelva punteros o referencias a variables locales.

El **constructor de copia** representa a un tipo especial de sobrecarga del constructor y se utiliza para gestionar de manera adecuada la copia de objetos. Como se ha discutido anteriormente, la copia exacta de objetos puede producir efectos no deseables, especialmente cuando se trata con asignación dinámica de memoria en el propio constructor.

Recuerde que C++ contempla dos tipos de situaciones distintas en las que el valor de un objeto se da a otro: la asignación y la inicialización. Esta segunda se pueda dar de tres formas distintas:

- Cuando un objeto inicializa explícitamente a otro, por ejemplo en una declaración.
- Cuando la copia de un objeto se pasa como parámetro en una función.

- Cuando se genera un objeto temporal, por ejemplo al devolverlo en una función.

Es importante resaltar que el constructor de copia sólo se aplica en las inicializaciones y no en las asignaciones. El siguiente listado de código muestra un ejemplo de implementación del constructor de copia, en el que se gestiona adecuadamente la asignación de memoria dinámica en el constructor. Dicho ejemplo se basa en el discutido anteriormente sobre el uso de paso de objetos por valor.

Como se puede apreciar, el constructor normal hace una reserva dinámica de memoria. Por lo tanto, el constructor de copia se utiliza para que, al hacer la copia, se reserve nueva memoria y se asigne el valor adecuado a la variable de clase.

Listado 3.15: Uso del constructor de copia

```
1  #include <iostream>
2  using namespace std;
3
4  class A {
5    int *_valor;
6  public:
7    A(int valor);     // Constructor.
8    A(const A &obj); // Constructor de copia.
9    ~A();            // Destructor.
10
11   int getValor () const { return *_valor;}
12 };
13
14 A::A (int valor) {
15   cout << "Construyendo..." << endl;
16   _valor = new int;
17   *_valor = valor;
18 }
19
20 A::A (const A &obj) {
21     cout << "Constructor copia..." << endl;
22   _valor = new int;
23   *_valor = obj.getValor();
24 }
25
26 A::~A () {
27   cout << "Destruyendo..." << endl;
28   delete _valor;
29 }
30
31 void mostrar (A a) {
32   cout << a.getValor() << endl;
33 }
34
35 int main () {
36   A a(7);
37   mostrar(a);
38   return 0;
39 }
```

La salida del programa es la siguiente:

```
Construyendo...
Constructor copia...
```

this y funciones amigas

Las funciones amigas no manejan el puntero *this*, debido a que no son miembros de una clase. Sólo las funciones miembro tienen acceso a *this*.

```
7
Destruyendo...
Destruyendo...
```

Finalmente, antes de pasar a la sección de sobrecarga de operadores, C++ contempla, al igual que en otros lenguajes de programación, el uso de **this** como el puntero al objeto que invoca una función miembro. Básicamente, dicho puntero es un parámetro implícito a todas las funciones miembro de una clase.

3.2.3. Sobrecarga de operadores

La sobrecarga de operadores permite definir de manera explícita el significado de un operador en relación a una clase. Por ejemplo, una clase que gestione la matriculación de alumnos podría hacer uso del operador + para incluir a un nuevo alumno. En C++, los operadores se pueden sobrecargar de acuerdo a los tipos de clase definidos por el usuario. De este modo, es posible integrar nuevos tipos de datos cuando sea necesario. La sobrecarga de operadores está muy relacionada con la sobrecarga de funciones, siendo necesario definir el significado del operador sobre una determinada clase.

El siguiente ejemplo muestra la sobrecarga del operador **+** en la clase *Point3D*.

Listado 3.16: Sobrecarga del operador +

```
1  class Point3D {
2  public:
3    Point3D ():
4      _x(0), _y(0), _z(0) {}
5    Point3D (int x, int y, int z):
6      _x(x), _y(y), _z(z) {}
7    Point3D operator+ (const Point3D &op2);
8
9  private:
10   int _x, _y, _z;
11 };
12
13 Point3D
14 Point3D::operator+
15 (const Point3D &op2) {
16   Point3D resultado;
17
18   resultado._x = this->_x + op2._x;
19   resultado._y = this->_y + op2._y;
20   resultado._z = this->_z + op2._z;
21
22   return resultado;
23 }
```

Como se puede apreciar, el operador + de la clase *Point3D* permite sumar una a una los distintos componentes vinculados a las variables miembro para, posteriormente, devolver el resultado. Es importante resaltar que, aunque la operación está compuesta de dos operandos, sólo se pasa un operando por parámetro. El segundo operando es implícito y se pasa mediante *this*.

Existen ciertas restricciones relativas a la sobrecarga de operadores: no es posible alterar la precedencia de cualquier operador, no es posible alterar el número de operandos requeridos por un operador y, en general, no es posible utilizar argumentos por defecto.

En C++, también es posible sobrecargar **operadores unarios**, como por ejemplo **++**. En este caso particular, no sería necesario pasar ningún parámetro de manera explícita, ya que la operación afectaría al parámetro implícito *this*.

Otro uso importante de la sobrecarga de operadores está relacionado con los problemas discutidos en la sección 3.2.2. C++ utiliza un constructor de copia por defecto que se basa en realizar una copia exacta del objeto cuando éste se pasa como parámetro a una función, cuando se devuelve de la misma o cuando se inicializa. Si el constructor de una clase realiza una reserva de recursos, entonces el uso implícito del constructor de copia por defecto puede generar problemas. La solución, como se ha comentado anteriormente, es el constructor de copia.

Sin embargo, el constructor de copia sólo se utiliza en las inicializaciones y no en las asignaciones. En el caso de realizar una asignación, el objeto de la parte izquierda de la asignación recibe por defecto una copia exacta del objeto que se encuentra a la derecha de la misma. Esta situación puede causar problemas si, por ejemplo, el objeto realiza una reserva de memoria. Si, después de una asignación, un objeto altera o libera dicha memoria, el segundo objeto se ve afectado debido a que sigue haciendo uso de dicha memoria. La solución a este problema consiste en sobrecargar el **operador de asignación**.

El siguiente listado de código muestra la implementación de una clase en la que se reserva memoria en el constructor, en concreto, un array de caracteres.

Listado 3.17: Sobrecarga del operador de asignación

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class A {
  char *_valor;
public:
  A() {_valor = 0;}
  A(const A &obj);  // Constructor de copia.
  ~A() {if(_valor) delete [] _valor;
    cout << "Liberando..." << endl;}

  void mostrar () const {cout << _valor << endl;}
  void set (char *valor);
};

A::A(const A &obj) {
  _valor = new char[strlen(obj._valor) + 1];
  strcpy(_valor, obj._valor);
}

void A::set (char *valor) {
  delete [] _valor;
  _valor = new char[strlen(valor) + 1];
  strcpy(_valor, valor);
}
```

El constructor de copia se define entre las líneas [18] y [21], reservando una nueva región de memoria para el contenido de *_valor* y copiando el mismo en la variable miembro. Por otra parte, las líneas [23–26] muestran la implementación de la función *set*, que modifica el contenido de dicha variable miembro.

El siguiente listado muestra la implementación de la función *entrada*, que pide una cadena por teclado y devuelve un objeto que alberga la entrada proporcionada por el usuario.

En primer lugar, el programa se comporta adecuadamente cuando se llama a *entrada*, particularmente cuando se devuelve la copia del objeto *a*, utilizando el constructor de copia previamente definido. Sin embargo, el programará abortará abruptamente cuando el objeto devuelto por *entrada* se asigna a *obj* en la función principal. Recuerde que en este caso se efectua una copia idéntica. El problema reside en que *obj.valor* apunta a la misma dirección de memoria que el objeto temporal, y éste último se destruye después de volver desde *entrada*, por lo que *obj.valor* apunta a memoria que acaba de ser liberada. Además, *obj.valor* se vuelve a liberar al finalizar el programa.

Listado 3.18: Sobrecarga del operador de asignación (cont.)

```
1  A entrada () {
2    char entrada[80];
3    A a;
4
5    cout << "Introduzca texto... ";
6    cin >> entrada;
7
8    a.set(entrada);
9    return a;
10 }
11
12 int main () {
13   A obj;
14   obj = entrada(); // Fallo.
15   obj.mostrar();
16   return 0;
17 }
```

Para solucionar este problema se sobrecarga el operador de asignación de copia en la clase en cuestión, tal y como muestra el siguiente listado de código.

Listado 3.19: Sobrecarga del operador de asignación (cont.)

```
1  A& A::operator= (const A &obj) {
2    if (strlen(obj._valor) > strlen(_valor)) {
3      delete [] _valor;
4      _valor = new char[strlen(obj._valor) + 1];
5    }
6
7    strcpy(_valor, obj._valor);
8
9    return *this;
10 }
```

En la función anterior se comprueba si la variable miembro tiene suficiente memoria para albergar el objeto pasado como parámetro (línea [2]). Si no es así, libera memoria y reserva la que sea necesaria para, posteriormente, devolver la copia de manera adecuada.

Finalmente, resulta especialmente relevante destacar que C++ permite la sobrecarga de cualquier operador, a excepción de *new*, *delete*, ->, ->* y el operador coma, que requieren otro tipo de técnicas. El resto de operadores se sobrecargan del mismo modo que los discutidos en esta sección.

3.3. Herencia y polimorfismo

La **herencia** representa uno de los conceptos fundamentales de la POO debido a que permite la creación de jerarquías de elementos. De este modo, es posible crear una clase base que define los aspectos o características comunes de una serie de elementos relacionados. A continuación, esta clase se puede extender para que cada uno de estos elementos añada las características únicas que lo diferencian del resto.

En C++, las clases que son heredadas se definen como **clases base**, mientras que las clases que heredan se definen como **clases derivadas**. Una clase derivada puede ser a su vez clase base, posibilitando la generación de jerarquías de clases. Dichas jerarquías permiten la creación de cadenas en las que los eslabones están representados por clases individuales.

Por otra parte, el **polimorfismo** es el término utilizado para describir el proceso mediante el cual distintas implementaciones de una misma función se utilizan bajo un mismo nombre. De este modo, se garantiza un acceso uniforme a la funcionalidad, aunque las características propias de cada operación sean distintas.

Polimorfismo

El polimorfismo se suele definir como *una interfaz, múltiples métodos* y representa una de los aspectos clave de la POO.

En C++, el polimorfismo está soportado tanto en tiempo de compilación como en tiempo de ejecución. Por una parte, la sobrecarga de operadores y de funciones son ejemplos de polimorfismo en tiempo de compilación. Por otra parte, el uso de clases derivadas y de funciones virtuales posibilitan el polimorfismo en tiempo de ejecución.

3.3.1. Herencia

El siguiente listado de código muestra la clase base *Vehículo* que, desde un punto de vista general, define un medio de transporte por carretera. De hecho, sus variables miembro son el número de ruedas y el número de pasajeros.

Listado 3.20: Clase base *Vehículo*

```
1  class Vehiculo {
2    int _ruedas;    // Privado. No accesible en clases derivadas.
3    int _pasajeros;
4
5  public:
6    void setRuedas (int ruedas) {_ruedas = ruedas;}
7    int getRuedas () const {return _ruedas;}
8    void setPasajeros (int pasajeros) {_pasajeros = pasajeros;}
9    int getPasajeros () const {return _pasajeros;}
10 };
```

La clase base anterior se puede extender para definir coches con una nueva característica propia de los mismos, como se puede apreciar en el siguiente listado.

En este ejemplo no se han definido los constructores de manera intencionada para discutir el acceso a los miembros de la clase. Como se puede apreciar en la línea 5 del siguiente listado, la clase *Coche* hereda de la clase *Vehículo*, utilizando el operador **:**. La palabra reservada **public** delante de *Vehículo* determina el tipo de acceso. En este caso concreto, el uso de *public* implica que todos los miembros públicos de la

Herencia y acceso

El modificador de acceso cuando se usa herencia es opcional. Sin embargo, si éste se especifica ha de ser **public**, **protected** o **private**. Por defecto, su valor es *private* si la clase derivada es efectivamente una clase. Si la clase derivada es una estructura, entonces su valor por defecto es *public*.

Listado 3.21: Clase derivada *Coche*

```
#include <iostream>
#include "Vehiculo.cpp"
using namespace std;

class Coche : public Vehiculo {
  int _PMA;

public:
  void setPMA (int PMA) {_PMA = PMA;}
  int getPMA () const {return _PMA;}

  void mostrar () const {
    cout << "Ruedas: " << getRuedas() << endl;
    cout << "Pasajeros: " << getPasajeros() << endl;
    cout << "PMA: " << _PMA << endl;
  }
};
```

clase base serán también miembros públicos de la clase derivada. En otras palabras, el efecto que se produce equivale a que los miembros públicos de *Vehículo* se hubieran declarado dentro de *Coche*. Sin embargo, desde *Coche* no es posible acceder a los miembros privados de *Vehículo*, como por ejemplo a la variable *_ruedas*.

El caso contrario a la herencia pública es la *herencia privada*. En este caso, cuando la clase base se hereda con *private*, entonces todos los miembros públicos de la clase base se convierten en privados en la clase derivada.

Además de ser público o privado, un miembro de clase se puede definir como **protegido**. Del mismo modo, una clase base se puede heredar como protegida. Si un miembro se declara como protegido, dicho miembro no es accesible por elementos que no sean miembros de la clase salvo en una excepción. Dicha excepción consiste en heredar un miembro protegido, hecho que marca la diferencia entre *private* y *protected*. En esencia, los miembros protegidos de la clase base se convierten en miembros protegidos de la clase derivada. Desde otro punto de vista, los miembros protegidos son miembros privados de una clase base pero con la posibilidad de heredarlos y acceder a ellos por parte de una clase derivada. El siguiente listado de código muestra el uso de *protected*.

Listado 3.22: Clase derivada *Coche*. Acceso protegido

```
#include <iostream>
using namespace std;

class Vehiculo {
protected:
  int _ruedas;    // Accesibles en Coche.
  int _pasajeros;
  // ...
};

class Coche : protected Vehiculo {
  int _PMA;

public:
  // ...
  void mostrar () const {
    cout << "Ruedas: " << _ruedas << endl;
    cout << "Pasajeros: " << _pasajeros << endl;
```

```
19      cout << "PMA: " << _PMA << endl;
20    }
21  };
```

Otro caso particular que resulta relevante comentar se da cuando una clase base se hereda como privada. En este caso, los miembros protegidos se heredan como miembros privados en la clase protegida.

Si una clase base se hereda como protegida mediante el modificador de acceso *protected*, entonces todos los miembros públicos y protegidos de dicha clase se heredan como miembros protegidos en la clase derivada.

Constructores y destructores

Inicialización de objetos

El constructor de una clase debería inicializar idealmente todo el estado de los objetos instanciados. Utilice el constructor de la clase base cuando así sea necesario.

Cuando se hace uso de herencia y se definen constructores y/o destructores de clase, es importante conocer el orden en el que se ejecutan en el caso de la clase base y de la clase derivada, respectivamente. Básicamente, a la hora de construir un objeto de una clase derivada, primero se ejecuta el constructor de la clase base y, a continuación, el constructor de la derivada. En el caso de la destrucción de objetos, el orden se invierte, es decir, primero se ejecuta el destructor de la clase derivada y, a continuación, el de la clase base.

Otro aspecto relevante está vinculado al paso de parámetros al constructor de la clase base desde el constructor de la clase derivada. Para ello, simplemente se realiza una llamada al constructor de la clase base, pasando los argumentos que sean necesarios. Este planteamiento es similar al utilizado en Java mediante *super()*. Note que aunque una clase derivada no tenga variables miembro, en su constructor han de especificarse aquellos parámetros que se deseen utilizar para llamar al constructor de la clase base.

Recuerde que, al utilizar herencia, los constructores se ejecutan en el orden de su derivación mientras que los destructores se ejecutan en el orden inverso a la derivación.

3.3.2. Herencia múltiple

C++ posibilita la herencia múltiple, es decir, permite que una clase derivada herede de **dos o más clases base**. Para ejemplificar la potencia de la herencia múltiple, a continuación se plantea un problema que se abordará con distintos enfoques con el objetivo de obtener una buena solución de diseño [27].

Suponga que es necesario diseñar la clase *ObjetoJuego*, la cual se utilizará como clase base para distintas entidades en un juego, como los enemigos, las cámaras, los *items*, etc. En concreto, es necesario que todos los objetos del juego soporten funcionalidad relativa a la recepción de mensajes y, de manera simultánea, dichos objetos han de poder relacionarse como parte de una estructura de árbol.

El enfoque *todo en uno*

Una primera opción de diseño podría consistir en aplicar un enfoque *todo en uno*, es decir, implementar los requisitos previamente comentados en la propia clase *ObjetoJuego*, añadiendo la funcionalidad de recepción de mensajes y la posibilidad de enlazar el objeto en cualquier parte del árbol a la propia clase.

Aunque la simplicidad de esta aproximación es su principal ventaja, en general añadir todo lo que se necesita en una única clase no es la mejor decisión de diseño. Si se utiliza este enfoque para añadir más funcionalidad, la clase crecerá en tamaño y en complejidad cada vez que se integre un nuevo requisito funcional. Así, una clase base que resulta fundamental en el diseño de un juego se convertirá en un elemento difícil de utilizar y de mantener. En otras palabras, la simplicidad a corto plazo se transforma en complejidad a largo plazo.

Otro problema concreto con este enfoque es la duplicidad de código, ya que la clase *ObjetoJuego* puede no ser la única en recibir mensajes, por ejemplo. La clase *Jugador* podría necesitar recibir mensajes sin ser un tipo particular de la primera clase. En el caso de enlazar con una estructura de árbol se podría dar el mismo problema, ya que otros elementos del juego, como por ejemplo los nodos de una escena se podría organizar del mismo modo y haciendo uso del mismo tipo de estructura de árbol. En este contexto, copiar el código allí donde sea necesario no es una solución viable debido a que complica enormemente el mantenimiento y afecta de manera directa a la arquitectura del diseño.

Enfoque basado en agregación

Contenedores

La aplicación de un esquema basado en agregación, de manera que una clase contiene elementos relevantes vinculados a su funcionalidad, es en general un buen diseño.

La conclusión directa que se obtiene al reflexionar sobre el anterior enfoque es que resulta necesario diseñar sendas clases, *ReceptorMensajes* y *NodoArbol*, para representar la funcionalidad previamente discutida. La cuestión reside en cómo relacionar dichas clases con la clase *ObjetoJuego*.

Una opción inmediata podría ser la agregación, de manera que un objeto de la clase *ObjetoJuego* contuviera un objeto de la clase *ReceptorMensajes* y otro de la clase *NodoArbol*, respectivamente. Así, la clase *ObjetoJuego* sería responsable de proporcionar la funcionalidad necesaria para manejarlos en su propia interfaz. En términos generales, esta solución proporciona un gran nivel de reutilización sin incrementar de manera significativa la complejidad de las clases que se extienden de esta forma.

El siguiente listado de código muestra una posible implementación de este diseño.

La desventaja directa de este enfoque es la generación de un gran número de funciones que simplemente llaman a la función de una variable miembro, las cuales han de crearse y mantenerse. Si unimos este hecho a un cambio en su interfaz, el mantenimiento se complica aún más. Así mismo, se puede producir una sobrecarga en el número de llamadas a función, hecho que puede reducir el rendimiento de la aplicación.

Una posible solución a este problema consiste en exponer los propios objetos en lugar de envolverlos con llamadas a funciones miembro. Este planteamiento simplifica el mantenimiento pero tiene la desventaja de que proporciona más información de la realmente necesaria en la clase *ObjetoJuego*. Si además, posteriormente, es necesario modificar la implementación de dicha clase con propósitos de incrementar la eficiencia, entonces habría que modificar todo el código que haga uso de la misma.

Uso de la herencia

Recuerde utilizar la herencia con prudencia. Un buen truco consiste en preguntarse si la clase derivada **es un** tipo particular de la clase base.

Listado 3.23: Clase *ObjetoJuego*. Uso agregación

```
1  class ObjetoJuego {
2  public:
3    bool recibirMensaje (const Mensaje &m);
4
5    ObjetoJuego* getPadre ();
6    ObjetoJuego* getPrimerHijo ();
7    // ...
8
9  private:
10   ReceptorMensajes *_receptorMensajes;
11   NodoArbol *_nodoArbol;
12 };
13
14 inline bool recibirMensaje (const Mensaje &m) {
15   return _receptorMensajes->recibirMensaje(m);
16 }
17
18 inline ObjetoJuego* getPadre () {
19   return _nodoArbol->getPadre();
20 }
21
22 inline ObjetoJuego* getPrimerHijo () {
23   return _nodoArbol->getPrimerHijo();
24 }
```

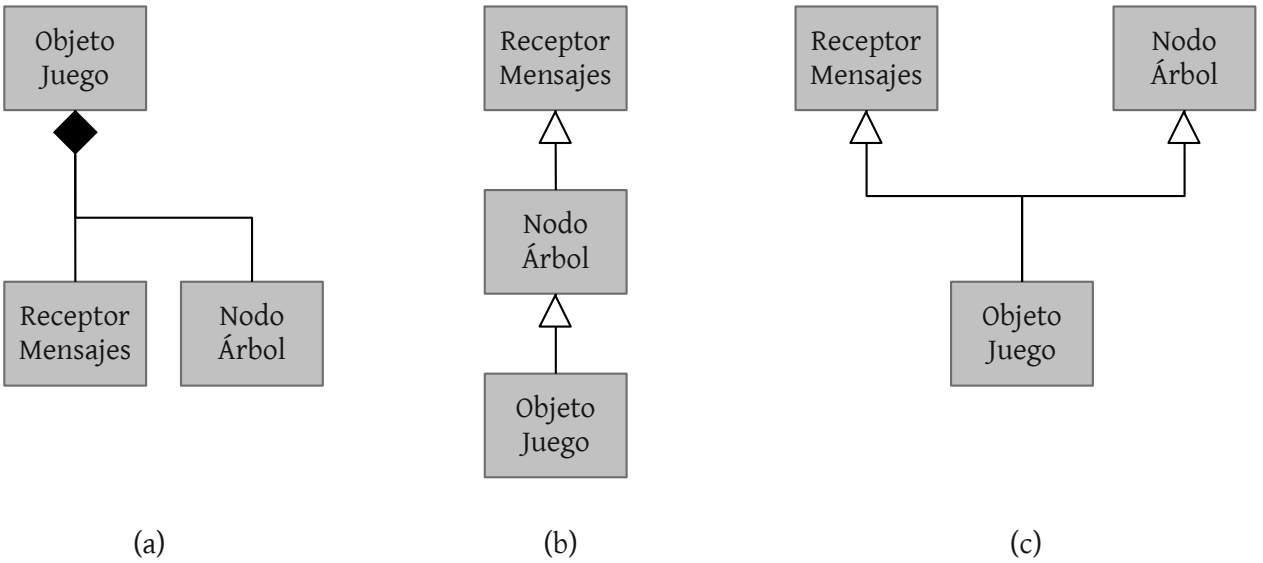


Figura 3.2: Distintas soluciones de diseño para el problema de la clase *ObjetoJuego*. **(a)** Uso de agregación. **(b)** Herencia simple. **(c)** Herencia múltiple.

Enfoque basado en herencia simple

Otra posible solución de diseño consiste en usar herencia simple, es decir, *ObjetoJuego* se podría declarar como una clase derivada de *ReceptorMensajes*, aunque *NodoArbol* quedaría aislado. Si se utiliza herencia simple, entonces una alternativa sería aplicar una cadena de herencia, de manera que, por ejemplo, *ArbolNodo* hereda de *ReceptorMensajes* y, a su vez, *ObjetoJuego* hereda de *ArbolNodo* (ver figura 3.2.b).

Aunque este planteamiento es perfectamente funcional, el diseño no es adecuado ya que resulta bastante lógico pensar que *ArbolNodo* no es un tipo especial de *ReceptorMensajes*. Si no es así, entonces no debería utilizarse herencia. Simple y llanamente. Del mismo modo, la relación inversa tampoco es lógica.

Enfoque basado en herencia múltiple

La herencia múltiple representa la solución idónea al problema planteado. Realmente, la herencia múltiple funciona como la herencia simple, pero posibilita que una clase derivada herede de dos clases base. En este caso particular, *ObjetoJuego* podría heredar de *ReceptorMensajes* y de *ArbolNodo* simultáneamente. Así, la primera clase tendría de manera automática la interfaz, las variables miembro y la funcionalidad de las otras dos clases.

Listado 3.24: Clase *ObjetoJuego*. Herencia múltiple

```
1 class ObjetoJuego: public ReceptorMensajes, public NodoArbol {
2 public:
3   // Funcionalidad necesaria.
4 };
```

Desde un punto de vista general, la herencia múltiple puede introducir una serie de complicaciones y desventajas, entre las que destacan las siguientes:

- **Ambigüedad**, debido a que las clases base de las que hereda una clase derivada pueden mantener el mismo nombre para una función. Para solucionar este problema, se puede explicitar el nombre de la clase base antes de hacer uso de la función, es decir, *ClaseBase::Funcion*.
- **Topografía**, debido a que se puede dar la situación en la que una clase derivada herede de dos clases base, que a su vez heredan de otra clase, compartiendo todas ellas la misma clase. Este tipo de árboles de herencia puede generar consecuencias inesperadas, como duplicidad de variables y ambigüedad. Este tipo de problemas se puede solventar mediante *herencia virtual*, concepto distinto al que se estudiará en el siguiente apartado relativo al uso de *funciones virtuales.*
- **Arquitectura del programa**, debido a que el uso de la herencia, simple o múltiple, puede contribuir a degradar el diseño del programa y crear un fuerte acoplamiento entre las distintas clases que la componen. En general, es recomendable utilizar alternativas como la composición y relegar el uso de la herencia múltiple sólo cuando sea la mejor alternativa real.

Las jerarquías de herencia que forman un diamante, conocidas comúnmente como DOD (Diamond Of Death) deberían evitarse y, generalmente, es un signo de un diseño incorrecto.

3.3.3. Funciones virtuales y polimorfismo

Como se introdujo al principio de la sección 3.3, en C++ el polimorfismo está soportado tanto en tiempo de compilación como en tiempo de ejecución. La sobrecarga de operadores y de funciones son ejemplos de polimorfismo en tiempo de compilación, mientras que el uso de clases derivadas y de funciones virtuales posibilitan el polimorfismo en tiempo de ejecución. Antes de pasar a discutir las funciones virtuales, se introducirá el concepto de puntero a la clase base como fundamento básico del polimorfismo en tiempo de ejecución.

Flexibilidad en C++

El polimorfismo en C++ es un arma muy poderosa y, junto con la herencia, permite diseñar e implementar programas complejos.

El puntero a la clase base

En términos generales, un puntero de un determinado tipo no puede apuntar a un objeto de otro tipo distinto. Sin embargo, los punteros a las clases bases y derivadas representan la excepción a esta regla. En C++, un puntero de una determinada clase base se puede utilizar para apuntar a objetos de una clase derivada a partir de dicha clase base.

En el listado de código 3.25 se retoma el ejemplo de uso de herencia entre las clases *Vehículo* y *Coche* para mostrar el uso de un puntero a una clase base (*Vehículo*). Como se puede apreciar en la línea (11), el puntero de tipo base *Vehículo* se utiliza para apuntar a un objeto de tipo derivado *Coche*, para, posteriormente, acceder al número de pasajeros en la línea (12).

Listado 3.25: Manejo de punteros a clase base

```
1  #include "Coche.cpp"
2
3  int main () {
4    Vehiculo *v; // Puntero a objeto de tipo vehículo.
5    Coche c;     // Objeto de tipo coche.
6
7    c.setRuedas(4);    // Se establece el estado de c.
8    c.setPasajeros(7);
9    c.setPMA(1885);
10
11   v = &c;      // v apunta a un objeto de tipo coche.
12   cout << "Pasajeros: " << v->getPasajeros() << endl;
13
14   return 0;
15 }
```

Cuando se utilizan punteros a la clase base, es importante recordar que sólo es posible acceder a aquellos elementos que pertenecen a la clase base. En el listado de código 3.26 se ejemplifica cómo no es posible acceder a elementos de una clase derivada utilizando un puntero a la clase base.

Fíjese cómo en la línea (13) el programa está intentando acceder a un elemento particular de la clase *Coche* mediante un puntero de tipo *Vehículo*. Obviamente, el compilador generará un error ya que la función *getPMA()* es específica de la clase derivada. Si se desea acceder a los elementos de una clase derivada a través de un puntero a la clase base, entonces es necesario utilizar un **molde** o *cast*.

En la línea (15) se muestra cómo realizar un *casting* para poder utilizar la funcionalidad anteriormente mencionada. Sin embargo, y aunque la instrucción es perfectamente válida, es preferible utilizar la nomenclatura de la línea (17), la cual es más limpia y hace uso de elementos típicos de C++.

Listado 3.26: Manejo de punteros a clase base (cont.)

```
#include "Coche.cpp"

int main () {
  Vehiculo *v; // Puntero a objeto de tipo vehículo.
  Coche c;     // Objeto de tipo coche.

  c.setRuedas(4);    // Se establece el estado de c.
  c.setPasajeros(7);
  c.setPMA(1885);

  v = &c;      // v apunta a un objeto de tipo coche.

  cout << v->getPMA() << endl; // ERROR en tiempo de compilación.

  cout << ((Coche*)v)->getPMA() << endl; // NO recomendable.

  cout << static_cast<Coche*>(v)->getPMA() << endl; // Estilo C++.

  return 0;
}
```

Otro punto a destacar está relacionado con la aritmética de punteros. En esencia, los punteros se incrementan o decrementan de acuerdo a la clase base. En otras palabras, cuando un puntero a una clase base está apuntando a una clase derivada y dicho puntero se incrementa, entonces no apuntará al siguiente objeto de la clase derivada. Por el contrario, apuntará a lo que *él cree que es* el siguiente objeto de la clase base. Por lo tanto, no es correcto incrementar un puntero de clase base que apunta a un objeto de clase derivada.

Finalmente, es importante destacar que, al igual que ocurre con los punteros, una **referencia** a una clase base se puede utilizar para referenciar a un objeto de la clase derivada. La aplicación directa de este planteamiento se da en los parámetros de una función.

Uso de funciones virtuales

La palabra clave *virtual*

Una clase que incluya una función virtual se denomina clase polimórfica.

Una función virtual es una función declarada como *virtual* en la clase base y redefinida en una o más clases derivadas. De este modo, cada clase derivada puede tener su propia versión de dicha función. El aspecto interesante es lo que ocurre cuando se llama a esta función con un puntero o referencia a la clase base. En este contexto, C++ determina en tiempo de ejecución qué versión de la función se ha de ejecutar en función del tipo de objeto al que apunta el puntero.

Listado 3.27: Uso básico de funciones virtuales

```
#include <iostream>
using namespace std;

class Base {
public:
  virtual void imprimir () const { cout << "Soy Base!" << endl; }
};

class Derivada1 : public Base {
public:
```

```
11   void imprimir () const { cout << "Soy Derivada1!" << endl; }
12 };
13
14 class Derivada2 : public Base {
15 public:
16   void imprimir () const { cout << "Soy Derivada2!" << endl; }
17 };
18
19 int main () {
20   Base *pb, base_obj;
21   Derivada1 d1_obj;
22   Derivada2 d2_obj;
23
24   pb = &base_obj;
25   pb->imprimir(); // Acceso a imprimir de Base.
26   pb = &d1_obj;
27   pb->imprimir(); // Acceso a imprimir de Derivada1.
28   pb = &d2_obj;
29   pb->imprimir(); // Acceso a imprimir de Derivada2.
30
31   return 0;
32 }
```

La ejecución del programa produce la siguiente salida:

```
Soy Base!
Soy Derivada1!
Soy Derivada2!
```

Sobrecarga/sobreescrit.

Cuando una función virtual se redefine en una clase derivada, la función se sobreescribe. Para sobrecargar una función, recuerde que el número de parámetros y/o sus tipos han de ser diferentes.

Las funciones virtuales han de ser miembros de la clase en la que se definen, es decir, no pueden ser funciones amigas. Sin embargo, una función virtual puede ser amiga de otra clase. Además, los destructores se pueden definir como funciones virtuales, mientras que en los constructores no es posible.

Las funciones virtuales se heredan de manera independiente del número de niveles que tenga la jerarquía de clases. Suponga que en el ejemplo anterior *Derivada2* hereda de *Derivada1* en lugar de heredar de *Base*. En este caso, la función *imprimir* seguiría siendo virtual y C++ sería capaz de seleccionar la versión adecuada al llamar a dicha función. Si una clase derivada no sobreescribe una función virtual definida en la clase base, entonces se utiliza la versión de la clase base.

El polimorfismo permite manejar la complejidad de los programas, garantizando la escalabilidad de los mismos, debido a que se basa en el principio de ***una interfaz, múltiples métodos***. Por ejemplo, si un programa está bien diseñado, entonces se puede suponer que todos los objetos que derivan de una clase base se acceden de la misma forma, incluso si las acciones específicas varían de una clase derivada a la siguiente. Esto implica que sólo es necesario recordar una interfaz. Sin embargo, la clase derivada es libre de añadir uno o todos los aspectos funcionales especificados en la clase base.

Es importante destacar que un aspecto clave para entender el polimorfismo reside en que la clase base y las derivadas forman una jerarquía, la cual plantea una evolución desde los aspectos más generales (clase base) hasta los aspectos más específicos (clases derivadas). Por lo tanto, diseñar correctamente la clase base es esencial, ya que define tanto los aspectos generales como aquellos aspectos que las clases derivadas tendrán que especificar.

Funciones virtuales puras y clases abstractas

Si una función virtual no se redefine en la clase derivada, entonces se utiliza la función definida en la clase base. Sin embargo, en determinadas situaciones no tiene sentido definir una función virtual en una clase base debido a que semánticamente no es correcto. El escenario típico se produce cuando existe una clase base, asociada a un concepto abstracto, para la que no pueden existir objetos. Por ejemplo, una clase *Figura* sólo tiene sentido como base de alguna clase derivada.

En estos casos, es posible implementar las funciones virtuales de la clase base de manera que generen un error, ya que su ejecución carece de sentido en este tipo de clases que manejan conceptos abstractos. Sin embargo, C++ proporciona un mecanismo para tratar este tipo de situaciones: el uso de *funciones virtuales puras*. Este tipo de funciones virtuales permite la definición de *clases abstractas*, es decir, clases a partir de las cuales no es posible instanciar objetos.

El siguiente listado de código muestra un ejemplo en el que se define la clase abstracta *Figura*, como base para la definición de figuras concretos, como el círculo. Note como la transformación de una función virtual en pura se consigue mediante el especificador **=0**.

Fundamentos POO

La encapsulación, la herencia y el polimorfimos representan los pilares fundamentales de la programación orientada a objetos.

Recuerde que una clase con una o más funciones virtuales puras es una clase abstracta y, por lo tanto, no se pueden realizar instancias a partir de ella. En realidad, la clase abstracta define una interfaz que sirve como **contrato funcional** para el resto de clases que hereden a partir de la misma. En el ejemplo anterior, la clase *Circulo* está obligada a implementar la función *area* en caso de definirla. En caso contrario, el compilador generará un error. De hecho, si una función virtual pura no se define en una clase derivada, entonces dicha función virtual sigue siendo pura y, por lo tanto, la clase derivada es también una clase abstracta.

Listado 3.28: Uso básico de funciones virtuales puras

```
1  #include <iostream>
2  using namespace std;
3
4  class Figura { // Clase abstracta Figura.
5  public:
6    virtual float area () const = 0; // Función virtual pura.
7  };
8
9  class Circulo : public Figura {
10 public:
11   Circulo (float r): _radio(r) {}
12   void setRadio (float r) { _radio = r; }
13   float getRadio () const { return _radio; }
14
15 // Redefinición de area () en Círculo.
16   float area () const { return _radio * _radio * 3.14; }
17
18 private:
19   float _radio;
20 };
21
22 int main () {
23   Figura *f;
24   Circulo c(1.0);
25
26   f = &c;
27   cout << "AREA: " << f->area() << endl;
28
29   // Recuerde realizar un casting al acceder a func. específica.
30   cout << "Radio:" << static_cast<Circulo*>(f)->getRadio() << endl;
31
32   return 0;
```

```
33 }
```

El uso más relevante de las clases abstractas consiste en proporcionar una interfaz sin revelar ningún aspecto de la implementación subyacente. Esta idea está fuertemente relacionada con la encapsulación, otro de los conceptos fundamentales de la POO junto con la herencia y el polimorfismo.

3.4. Plantillas

En el desarrollo de software es bastante común encontrarse con situaciones en las que los programas implementados se parecen enormemente a otros implementados con anterioridad, salvo por la necesidad de tratar con distintos tipos de datos o de clases. Por ejemplo, un mismo algoritmo puede mantener el mismo comportamiento de manera que éste no se ve afectado por el tipo de datos a manejar.

En esta sección se discute el uso de las plantillas en C++, un mecanismo que permite escribir **código genérico** sin tener dependencias explícitas respecto a tipos de datos específicos.

3.4.1. Caso de estudio. Listas

En el ámbito del desarrollo de videojuegos, una de las principales estructuras de datos manejadas es la lista de elementos. Por ejemplo, puede existir una lista que almacene las distintas entidades de nuestro juego, otra lista que contenga la lista de mallas poligonales de un objeto o incluso es bastante común disponer de listas que almacenen los nombres de los jugadores en el modo multijugador. Debido a que existe una fuerte necesidad de manejar listas con distintos tipos de datos, es importante plantear una implementación que sea mantenible y práctica para tratar con esta problemática.

General y óptimo

Recuerde que en el desarrollo de videojuegos siempre existe un compromiso entre plantear una solución general y una solución optimizada para la plataforma sobre la que se ejecutará el juego en cuestión.

Una posible alternativa consiste en que la propia clase que define los objetos contenidos en la lista actúe como **propio nodo** de la misma, es decir, que la propia clase sirva para implementar la lista (ver figura 3.3.a). Para ello, simplemente hay que mantener un enlace al siguiente elemento, o dos enlaces si se pretende construir una lista doblemente enlazada. El siguiente listado de código muestra un ejemplo de implementación.

Listado 3.29: Implementación de listas con nodos enlace

```
1  class Entidad {
2  public:
3    // Funcionalidad de la lista.
4    Entidad * getSiguiente ();
5    void eliminar ();
6    void insertar (Entidad *pNuevo);
7
8  private:
9    // Puntero a la cabeza de la lista.
10   Entidad *_pSiguiente;
11 };
```

Aunque este planteamiento es muy sencillo y funciona correctamente, la realidad es que adolece de varios problemas:

- Es propenso a errores de programación. El desarrollador ha de recordar casos particulares en la implementación, como por ejemplo la eliminación del último elemento de la lista.

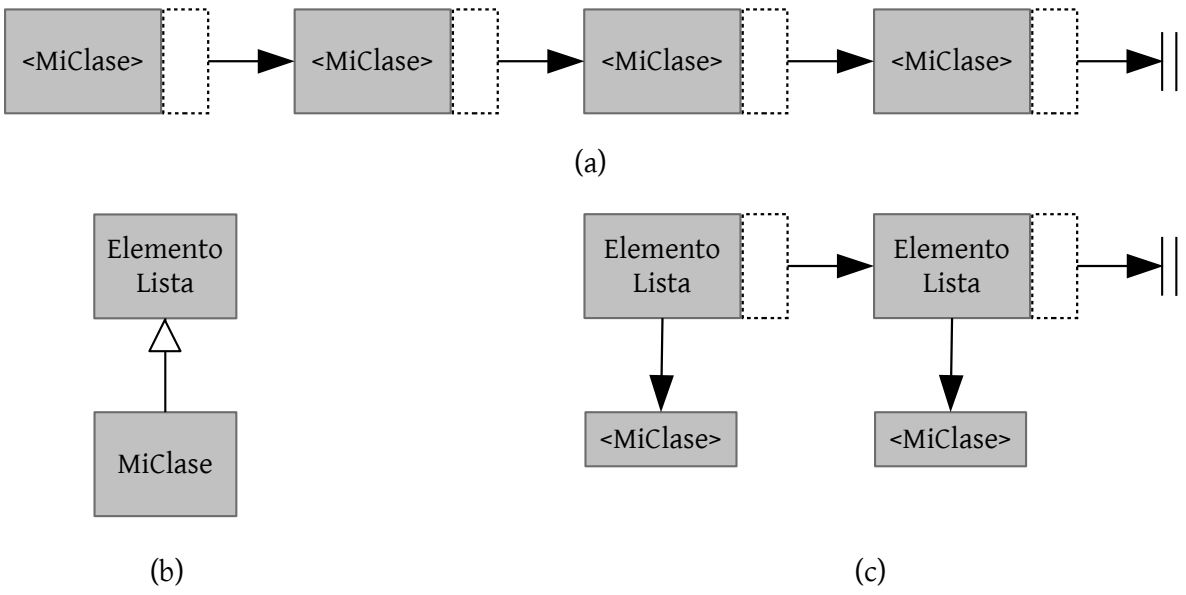


Figura 3.3: Distintos enfoques para la implementación de una lista con elementos génericos. **(a)** Integración en la propia clase de dominio. **(b)** Uso de herencia. **(c)** Contenedor con elementos de tipo nulo.

- Un cambio en la implementación de la clase previamente expuesta implicaría cambiar un elevado número de clases.
- No es correcto suponer que todas las listas manejadas en nuestro programa van a tener la misma interfaz, es decir, la misma funcionalidad. Además, es bastante probable que un desarrollador utilice una nomenclatura distinta a la hora de implementar dicha interfaz.

Otra posible solución consiste en hacer uso de la **herencia** para definir una clase base que represente a cualquier elemento de una lista (ver figura 3.3.b). De este modo, cualquier clase que desee incluir la funcionalidad asociada a la lista simplemente ha de extenderla. Este planteamiento permite tratar a los elementos de la lista mediante polimorfismo. Sin embargo, la mayor desventaja que presenta este enfoque está en el diseño, ya que no es posible separar la funcionalidad de la lista de la clase propiamente dicha, de manera que no es posible tener el objeto en múltiples listas o en alguna otra estructura de datos. Por lo tanto, es importante separar la propia lista de los elementos que realmente contiene.

Una alternativa para proporcionar esta separación consiste en hacer uso de una lista que maneja **punteros de tipo nulo** para albergar distintos tipos de datos (ver figura 3.3.c). De este modo, y mediante los moldes correspondientes, es posible tener una lista con elementos de distinto tipo y, al mismo tiempo, la funcionalidad de la misma está separada del contenido. La principal desventaja de esta aproximación es que no es *type-safe*, es decir, depende del programador incluir la funcionalidad necesaria para convertir tipos, ya que el compilador no los detectará.

Otra desventaja de esta propuesta es que son necesarias dos reservas de memoria para cada uno de los nodos de la lista: una para el objeto y otra para el siguiente nodo de la lista. Este tipo de cuestiones han de considerarse de manera especial en el desarrollo de videojuegos, ya que la plataforma hardware final puede tener ciertas restricciones de recursos.

La figura 3.3 muestra de manera gráfica las distintas opciones discutidas hasta ahora en lo relativo a la implementación de una lista que permita el tratamiento de datos genéricos.

3.4.2. Utilizando plantillas en C++

C++ proporciona el concepto de plantilla como solución a la implementación de código genérico, es decir, código que no esté vinculado a ningún tipo de datos en particular. La idea principal reside en instanciar la plantilla para utilizarla con un tipo de datos en particular. Existen dos tipos principales de plantillas: i) plantillas de clases y ii) plantillas de funciones.

Las **plantillas de clases** permiten utilizar tipos de datos genéricos asociados a una clase, posibilitando posteriormente su instanciación con tipos específicos. El siguiente listado de código muestra un ejemplo muy sencillo. Como se puede apreciar, se ha definido una clase *Triángulo* que se puede utilizar para almacenar cualquier tipo de datos. En este ejemplo, se ha instanciado un triángulo con elementos del tipo *Vec2*, que representan valores en el espacio bidimensional. Para ello, se ha utilizado la palabra clave **template** para completar la definición de la clase *Triángulo*, permitiendo el manejo de tipos genéricos **T**. Note cómo este tipo genérico se usa para declarar las variables miembro de dicha clase.

Clase *Vec2*

La clase *Vec2* se podría haber extendido mediante el uso de plantillas para manejar otros tipos de datos comunes a la representación de puntos en el espacio bidimensional, como por ejemplo valores en punto flotante.

Listado 3.30: Implementación de un triángulo con plantilla

```
1  #include <iostream>
2  using namespace std;
3
4  template<class T> // Tipo general T.
5  class Triangle {
6  public:
7    Triangle (const T &v1, const T &v2, const T &v3):
8      _v1(v1), _v2(v2), _v3(v3) {}
9    ~Triangle () {}
10   T getV1 () const { return _v1; }
11   T getV2 () const { return _v2; }
12   T getV3 () const { return _v3; }
13
14 private:
15   T _v1, _v2, _v3;
16 };
17
18 class Vec2 {
19 public:
20   Vec2 (int x, int y): _x(x), _y(y) {}
21   ~Vec2 () {}
22   int getX () const { return _x; }
23   int getY () const { return _y; }
24
25 private:
26   int _x, _y;
27 };
28
29 int main () {
30   Vec2 p1(2, 7), p2(3, 4), p3(7, 10);
31   Triangle<Vec2> t(p1, p2, p3); // Instancia de la plantilla.
32
33   cout << "V1: [" << t.getV1().getX() << ", "
34        << t.getV1().getY() << "]" << endl;
35   return 0;
36 }
```

Las **plantillas de funciones** siguen la misma idea que las plantillas de clases aplicándolas a las funciones. Obviamente, la principal diferencia con respecto a las plantillas a clases es que no necesitan instanciarse. El siguiente listado de código muestra un ejemplo sencillo de la clásica función *swap* para intercambiar el contenido de dos variables. Dicha función puede utilizarse con enteros, valores en punto flotante, cadenas o cualquier clase con un constructor de copia y un constructor de asignación. Además, la función se instancia dependiendo del tipo de datos utilizado. Recuerde que no es posible utilizar dos tipos de datos distintos, es decir, por ejemplo un entero y un valor en punto flotante, ya que se producirá un error en tiempo de compilación.

Listado 3.31: Ejemplo de uso plantillas de funciones

```
1  #include <iostream>
2  using namespace std;
3
4  template<class T> // Tipo general T.
5  void swap (T &a, T &b) {
6    T aux(a);
7    a = b;
8    b = aux;
9  }
10
11 int main () {
12   string a = "Hello", b = "Good-bye";
13   cout << "[" << a << ", " << b << "]" << endl;
14
15   swap(a, b); // Se instancia para cadenas.
16
17   cout << "[" << a << ", " << b << "]" << endl;
18   return 0;
19 }
```

Uso de plantillas

Las plantillas son una herramienta excelente para escribir código que no dependa de un tipo de datos específico.

El uso de plantillas en C++ solventa todas las necesidades planteadas para manejar las listas introducidas en la sección 3.4.1, principalmente las siguientes:

- **Flexibilidad**, para poder utilizar las listas con distintos tipos de datos.
- **Simplicidad**, para evitar la copia de código cada vez que se utilice una estructura de lista.
- **Uniformidad**, ya que se maneja una única interfaz para la lista.
- **Independencia**, entre el código asociado a la funcionalidad de la lista y el código asociado al tipo de datos que contendrá la lista.

A continuación se muestra la implementación de una posible solución, la cual está compuesta por dos clases generales. La primera de ellas se utilizará para los nodos de la lista y la segunda para especificar la funcionalidad de la propia lista.

Como se puede apreciar en el siguiente listado de código, las dos clases están definidas para poder utilizar cualquier tipo de dato y, además, la funcionalidad de la lista es totalmente independiente del su contenido.

Listado 3.32: Uso de plantillas para implementar listas

```
1  template<class T>
2  class NodoLista {
3  public:
4    NodoLista (T datos);
5    T & getDatos ();
6    NodoLista * siguiente ();
```

```
 7
 8 private:
 9   T _datos;
10 };
11
12 template<class T>
13 class Lista {
14 public:
15   NodoLista<T> getCabeza ();
16   void insertarFinal (T datos);
17   // Resto funcionalidad...
18
19 private:
20   NodoLista<T> *_cabeza;
21 };
```

3.4.3. ¿Cuándo utilizar plantillas?

Las plantillas de C++ representan un arma muy poderosa para implementar código génerico que se pueda utilizar con distintos tipos de datos. Sin embargo, al igual que ocurre con la herencia, hay que ser especialmente cuidadoso a la hora de utilizarlas, debido a que un uso inadecuado puede generar problemas y retrasos en el desarrollo de software. A continuación se enumeran las principales desventajas que plantea el uso de plantillas [27]:

Equipo de desarrollo

Es importante recordar la experiencia de los compañeros, actuales y futuros, en un grupo de desarrollo a la hora de introducir dependencias con aspectos avanzados en el uso de plantillas en C++.

- **Complejidad**, debido a la integración de nueva nomenclatura que puede dificultar la legibilidad del código. Además, el uso de plantillas hace que la depuración de código sea más difícil.
- **Dependencia**, ya que el código de la plantilla ha de incluirse en un fichero de cabecera para que sea visible por el compilador a la hora de instanciarlo. Este planteamiento incrementa el acoplamiento entre clases. Además, el tiempo de compilación se ve afectado.
- **Duplicidad de código**, debido a que si, por ejemplo, se crea una lista con un nuevo tipo de datos, el compilador ha de crear una nueva clase de lista. Es decir, todas las funciones y variables miembro se duplican. En el desarrollo de videojuegos, este inconveniente es generalmente asumible debido a la magnitud de los proyectos.
- **Soporte del compilador**, ya que las plantillas no existen como una solución plenamente estandarizada, por lo que es posible, aunque poco probable, que algunos compiladores no las soporten.

Desde una perspectiva general, no debe olvidar que las plantillas representan una herramienta adecuada para un determinado uso, por lo que su uso indiscriminado es un error. Recuerde también que las plantillas introducen una dependencia de uso respecto a otras clases y, por lo tanto, su diseño debería ser simple y mantenible.

Una de las situaciones en las que el uso de plantillas resulta adecuado está asociada al uso de **contenedores**, es decir, estructuras de datos que contienen objetos de distintas clases. En este contexto, es importante destacar que la biblioteca STL de C++ ya proporciona una implementación de listas, además de otras estructuras de datos y de algoritmos listos para utilizarse. Por lo tanto, es bastante probable que el desarrollador haga un uso directo de las mismas en lugar de tener que desarrollar desde cero su propia implementación. En el capítulo 5 se estudia el uso de la biblioteca STL y se discute su uso en el ámbito del desarrollo de videojuegos.

3.5. Manejo de excepciones

Freezing issues

Aunque el desarrollo de videojuegos comerciales madura año a año, aún hoy en día es bastante común encontrar errores y *bugs* en los mismos. Algunos de ellos obligan incluso a *resetear* la estación de juegos por completo.

A la hora de afrontar cualquier desarrollo software, un programador siempre tiene que tratar con los errores que dicho software puede generar. Existen diversas estrategias para afrontar este problema, desde simplemente ignorarlos hasta hacer uso de técnicas que los controlen de manera que sea posible recuperarse de los mismos. En esta sección se discute el manejo de excepciones en C++ con el objetivo de escribir **programas robustos** que permitan gestionar de manera adecuada el tratamiento de errores y situaciones inesperadas. Sin embargo, antes de profundizar en este aspecto se introducirán brevemente las distintas alternativas más relevantes a la hora de tratar con errores.

3.5.1. Alternativas existentes

La estrategia más simple relativa al tratamiento de errores consiste en **ignorarlos**. Aunque parezca una opción insensata y arriesgada, la realidad es que la mayoría de programas hacen uso de este planteamiento en una parte relevante de su código fuente. Este hecho se debe, principalmente, a que el programador asume que existen distintos tipos de errores que no se producirán nunca, o al menos que se producirán con una probabilidad muy baja. Por ejemplo, es bastante común encontrar la sentencia *fclose* sin ningún tipo de comprobación de errores, aún cuando la misma devuelve un valor entero indicando si se ha ejecutado correctamente o no.

En el caso particular del desarrollo de videojuegos, hay que ser especialmente cuidadoso con determinadas situaciones que en otros dominios de aplicación pueden no ser tan críticas. Por ejemplo, no es correcto suponer que un PC tendrá memoria suficiente para ejecutar un juego, siendo necesario el tratamiento explícito de situaciones como una posible falta de memoria por parte del sistema. Desde otro punto de vista, el usuario que compra un videojuego profesional asume que éste nunca va a fallar de manera independiente a cualquier tipo de situación que se pueda producir en el juego, por lo que el desarrollador de videojuegos está obligado a considerar de manera especial el tratamiento de errores.

Tradicionalmente, uno de los enfoques más utilizados ha sido el **retorno de códigos de error**, típico en lenguajes de programación de sistemas como C. Desde un punto de vista general, este planteamiento se basa en devolver un código numérico de error, o al menos un valor booleano, indicando si una función se ejecutó correctamente o no. El código que realiza la llamada a la función es el responsable de recoger y tratar dicho valor de retorno.

Este enfoque tiene su principal desventaja en el mantenimiento de código, motivado fundamentalmente por la necesidad de incluir bloques *if-then-else* para capturar y gestionar los posibles errores que se puedan producir en un fragmento de código. Además, la inclusión de este tipo de bloques complica la legibilidad del código, dificultando el entendimiento y ocultando el objetivo real del mismo. Finalmente, el rendimiento del programa también se ve afectado debido a que cada llamada que realizamos ha de estar envuelta en una sentencia *if*.

Constructores

El uso de códigos de error presenta una dificultad añadida en el uso de constructores, ya que estos no permiten la devolución de valores. En los destructores se da la misma situación.

Otra posible alternativa para afrontar el tratamiento de errores consiste en utilizar **aserciones (*asserts*)**, con el objetivo de parar la ejecución del programa y evitar así una posible terminación abrupta. Obviamente, en el desarrollo de videojuegos esta alternativa no es aceptable, pero sí se puede utilizar en la fase de depuración para obtener la mayor cantidad de información posible (por ejemplo, la línea de código que produjo el error) ante una situación inesperada.

Hasta ahora, los enfoques comentados tienen una serie de desventajas importantes. En este contexto, las **excepciones** se posicionan como una alternativa más adecuada y práctica. En esencia, el uso de excepciones permite que cuando un programa se tope con una situación inesperada se arroje una excepción. Este hecho tiene como consecuencia que el flujo de ejecución salte al bloque de captura de excepciones más cercano. Si dicho bloque no existe en la función en la que se arrojó la excepción, entonces el programa gestionará de manera adecuada la salida de dicha función (destruyendo los objetos vinculados) y saltará a la función padre con el objetivo de buscar un bloque de captura de excepciones.

Este proceso se realiza recursivamente hasta encontrar dicho bloque o llegará a la función principal delegando en el código de manejo de errores por defecto, que típicamente finalizará la ejecución del programa y mostrará información por la salida estándar o generará un fichero de *log*.

Los bloques de tratamiento de excepciones ofrecen al desarrollador la flexibilidad de hacer lo que desee con el error, ya sea ignorarlo, tratar de recuperarse del mismo o simplemente informar sobre lo que ha ocurrido. Este planteamiento facilita enormemente la distinción entre distintos tipos de errores y, consecuentemente, su tratamiento.

Recuerde que después de la ejecución de un bloque de captura de excepciones, la ejecución del programa continuará a partir de este punto y no desde donde la excepción fue arrojada.

Además de la mantenibilidad del código y de la flexibilidad del planteamiento, las excepciones permiten enviar más información sobre la naturaleza del error detectado a las capas de nivel superior. Por ejemplo, si se ha detectado un error al abrir un fichero, la interfaz gráfica sería capaz de especificar el fichero que generó el problema. Por el contrario, la utilización de códigos de error no permitiría manejar información más allá de la detección de un error de entrada/salida.

3.5.2. Excepciones en C++

El uso de excepciones en C++ es realmente sencillo y gira en torno a tres palabras clave: *throw, catch* y *try*. En resumen, cuando un fragmento de código necesita arrojar una excepción, entonces hace uso de **throw**. El control del programa pasa entonces al bloque de código de captura de excepciones más cercano, representado por la sentencia **catch**. Este bloque de captura está vinculado obligatoriamente a un bloque de código en el cual se podría lanzar la excepción, el cual está a su vez envuelto por una sentencia **try**.

El siguiente listado de código muestra un ejemplo muy sencillo de captura de excepciones (líneas 9–11) ante la posibilidad de que el sistema no pueda reservar memoria (líneas 6–8). En este caso particular, el programa captura una excepción definida en la biblioteca estándar de C++ denominada *bad_alloc*, de manera que se contempla un posible lanzamiento de la misma cuando se utiliza el operador *new* para asignar memoria de manera dinámica.

Excepciones estándar

C++ maneja una jerarquía de excepciones estándar para tipos generales, como por ejemplo *logic_error* o *runtime_error*, o aspectos más específicos, como por ejemplo *out_of_range* o *bad_alloc*. Algunas de las funciones de la biblioteca estándar de C++ lanzan algunas de estas excepciones.

Listado 3.33: Uso básico de excepciones

```
1 #include <iostream>
2 #include <exception>
3 using namespace std;
```

```
 4
 5 int main () {
 6   try {
 7     int *array = new int[1000000];
 8   }
 9   catch (bad_alloc &e) {
10     cerr << "Error al reservar memoria." << endl;
11   }
12
13   return 0;
14 }
```

Como se ha comentado anteriormente, la sentencia *throw* se utiliza para arrojar excepciones. C++ es estremadamente flexible y permite lanzar un objeto de cualquier tipo de datos como excepción. Este planteamiento posibilita la creación de **excepciones definidas por el usuario** que modelen las distintas situaciones de error que se pueden dar en un programa e incluyan la información más relevante vinculadas a las mismas.

El siguiente listado de código muestra un ejemplo de creación y tratamiento de excepciones definidas por el usuario.

En particular, el código define una excepción general en las líneas 4-10 mediante la definición de la clase *MiExcepcion*, que tiene como variable miembro una cadena de texto que se utilizará para indicar la razón de la excepción. En la función *main*, se lanza una instancia de dicha excepción, definida en la línea 21, cuando el usuario introduce un valor numérico que no esté en el rango [1, 10]. Posteriormente, dicha excepción se captura en las líneas 24-26.

Listado 3.34: Excepción definida por el usuario

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 class MiExcepcion {
 5   const string &_razon;
 6
 7 public:
 8   MiExcepcion (const string &razon): _razon(razon) {}
 9   const string &getRazon () const {return _razon;}
10 };
11
12 int main () {
13   int valor;
14   const string &r = "Valor introducido incorrecto.";
15
16   try {
17     cout << "Introduzca valor entre 1 y 10...";
18     cin >> valor;
19
20     if ((valor < 1) || (valor > 10)) {
21       throw MiExcepcion(r);
22     }
23   }
24   catch (MiExcepcion &e) {
25     cerr << e.getRazon() << endl;
26   }
27
28   return 0;
29 }
```

Excepciones y clases

Las excepciones se modelan exactamente igual que cualquier otro tipo de objeto y la definición de clase puede contener tanto variables como funciones miembro.

Normalmente, será necesario tratar con distintos tipos de excepciones de manera simultánea en un mismo programa. Un enfoque bastante común consiste en hacer uso de una jerarquía de excepciones, con el mismo planteamiento usado en una jerarquía de clases, para modelar distintos tipos de excepciones específicas.

La figura 3.4 muestra un ejemplo representativo vinculado con el desarrollo de videojuegos, en la que se plantea una jerarquía con una clase base y tres especializaciones asociadas a errores de gestión de memoria, E/S y operaciones matemáticas.

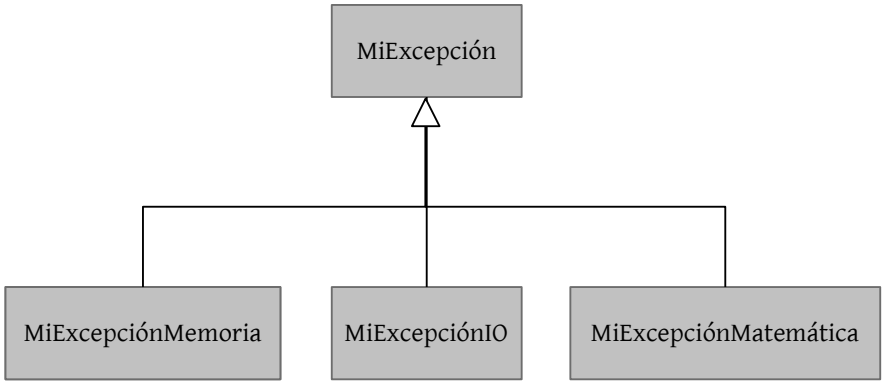


Figura 3.4: Ejemplo de jerarquía de excepciones.

Como se ha comentado anteriormente, los bloques de sentencias *try-catch* son realmente flexibles y posibilitan la gestión de diversos tipos de excepciones. El siguiente listado de código muestra un ejemplo en el que se carga información tridimensional en una estructura de datos.

Listado 3.35: Gestión de múltiples excepciones

```
1  void Mesh::cargar (const char *archivo) {
2    try {
3      Stream stream(archivo); // Puede generar un error de I/O.
4      cargar(stream);
5    }
6    catch (MiExcepcionIO &e) {
7      // Gestionar error I/O.
8    }
9    catch (MiExcepcionMatematica & e) {
10     // Gestionar error matemático.
11   }
12   catch (MiExcepcion &e) {
13     // Gestionar otro error...
14   }
15   catch (...) {
16     // Cualquier otro tipo de error...
17   }
18 }
```

La idea del anterior fragmento de código se puede resumir en que el desarrollador está preocupado especialmente por la gestión de errores de entrada/salida o mátematicos (primer y segundo bloque *catch*, respectivamente) pero, al mismo tiempo, no desea que otro tipo de error se propague a capas superiores, al menos un error que esté definido en la jerarquía de excepciones (tercer bloque *catch*). Finalmente, si se

desea que el programa capture cualquier tipo de excepción, entonces se puede añadir una captura genérica (ver cuarto bloque *catch*). En resumen, si se lanza una excepción no contemplada en un bloque *catch*, entonces el programa seguirá buscando el bloque *catch* más cercano.

Es importante resaltar que el **orden de las sentencias *catch*** es relevante, ya que dichas sentencias siempre se procesan de arriba a abajo. Además, cuando el programa encuentra un bloque que trata con la excepción lanzada, el resto de bloques se ignoran automáticamente.

Exception handlers

El tratamiento de excepciones se puede enfocar con un esquema parecido al del tratamiento de eventos, es decir, mediante un planteamiento basado en capas y que delege las excepciones para su posterior gestión.

Otro aspecto que permite C++ relativo al manejo de excepciones es la posibilidad de **re-lanzar** una excepción, con el objetivo de delegar en una capa superior el tratamiento de la misma. El siguiente listado de código muestra un ejemplo en el que se delega el tratamiento del error de entrada/salida.

Listado 3.36: Re-lanzando una excepción

```
1  void Mesh::cargar (const char *archivo) {
2
3    try {
4      Stream stream(archivo); // Puede generar un error de I/O.
5      cargar(stream);
6    }
7
8    catch (MiExcepcionIO &e) {
9      if (e.datosCorruptos()) {
10       // Tratar error I/O.
11     }
12     else {
13       throw; // Se re-lanza la excepción.
14     }
15   }
16
17 }
```

3.5.3. ¿Cómo manejar excepciones adecuadamente?

Además de conocer cómo utilizar las sentencias relativas al tratamiento de excepciones, un desarrollador ha de conocer cómo utilizarlas de manera adecuada para evitar problemas potenciales, como por ejemplo no liberar memoria que fue reservada previamente al lanzamiento de una excepción. En términos generales, este problema se puede extrapolar a cómo liberar un recurso que se adquirió previamente a la generación de un error.

El siguiente listado de código muestra cómo utilizar excepciones para liberar correctamente los **recursos previamente reservados** en una función relativa a la carga de texturas a partir de la ruta de una imagen.

Como se puede apreciar, en la función *cargarTextura* se reserva memoria para el manejador del archivo y para el propio objeto de tipo textura. Si se generase una excepción dentro del bloque *try*, entonces se ejecutaría el bloque *catch* genérico que se encarga de liberar los dos recursos previamente mencionados. Así mismo, todos los recursos *locales* de la propia función se destruirán tras salir de la misma.

Smart pointers

En C++ es bastante común utilizar herramientas que permitan manejar los punteros de una forma más cómodo. Un ejemplo representativo son los punteros *inteligentes* o *smart pointers*.

Sin embargo, este planteamiento se puede mejorar considerando especialmente la naturaleza de los recursos manejados en la función, es decir, los propios punteros. El resto de recursos utilizados en la función tendrán sus destructores correctamente implementados y se puede suponer que finalizarán adecuadamente tras la salida de la función en caso de que se genere una excepción. La parte problemática está representada por los **punteros**, ya que no tienen asociado destructores.

Listado 3.37: Uso adecuado de excepciones

```
1  Textura * cargarTextura (const char *ruta) {
2    FILE *entrada = NULL;
3    Textura *pTextura = NULL;
4
5    try {
6      entrada = fopen(ruta, "rb");
7      // Instanciar recursos locales...
8      pTextura = new Textura(/*..., ...*/);
9      leerTextura(entrada, pTextura);
10   }
11   catch (...) { // Liberar memoria ante un error.
12     delete pTextura;
13     pTexture = NULL;
14   }
15
16   fclose(entrada);
17   return pTextura;
18 }
```

En el caso del manejador de archivos, una solución elegante consiste en construir un *wrapper*, es decir, una clase que envuelva la funcionalidad de dicho manejador y que su constructor haga uso de *fopen* mientras que su destructor haga uso de *fclose*. Así, si se crea un objeto de ese tipo en la pila, el manejador del archivo se liberará cuando dicho objeto quede fuera de alcance (de la función).

En el caso del puntero a la textura, es posible aplicar una solución más sencilla para todos los punteros: el uso de la plantilla **unique_ptr**. Dicha clase forma parte del estándar de 2011 de C++ y permite plantear un enfoque similar al del anterior manejador pero con punteros en lugar de con ficheros. Básicamente, cuando un objeto de dicha clase se destruye, entonces el puntero asociado también se libera correctamente. A continuación se muestra un listado de código con las dos soluciones discutidas.

Listado 3.38: Uso adecuado de excepciones (simple)

```
1  unique_ptr<Textura> cargarTextura (const char *ruta) {
2    FilePtr entrada(ruta, "rb");
3    // Instanciar recursos locales...
4    unique_ptr<Textura> pTextura(new Textura(/*..., ...*/));
5
6    leerTextura(entrada, pTextura);
7    return pTextura;
8  }
```

Como se puede apreciar, no es necesario incluir ningún tipo de código de manejo de excepciones, ya que la propia función se encarga de manera implícita gracias al enfoque planteado. Recuerde que una vez que el flujo del programa abandone la función, ya sea de manera normal o provocado por una excepción, todo los recursos habrán sido liberados de manera adecuada.

Finalmente, es importante destacar que este tipo de punteros *inteligentes* se pueden utilizar en otro tipo de situaciones específicas, como por ejemplo los **constructores**. Recuerde que, si se genera una excepción en un constructor, no es posible devolver ningún código de error. Además, si ya se reservó memoria en el constructor, entonces

se producirá el clásico *memory leak*, es decir, la situación en la que no se libera una porción de memoria que fue previamente reservada. En estos casos, el destructor no se ejecuta ya que, después de todo, el constructor no finalizó correctamente su ejecución. La solución pasa por hacer uso de este tipo de punteros.

El caso de los **destructores** es menos problemático, ya que es lógico suponer que nadie hará uso de un objeto que se va a destruir, incluso cuando se genere una excepción dentro del propio destructor. Sin embargo, es importante recordar que el destructor no puede lanzar una excepción si el mismo fue llamado como consecuencia de otra excepción.

3.5.4. ¿Cuándo utilizar excepciones?

Plataformas HW

En el ámbito del desarrollo de videojuegos, el uso de excepciones está estrechamente con la plataforma HW final sobre la que se ejecutará el juego en cuestión, debido al impacto en el rendimiento que tienen las mismas.

En el ámbito particular del tratamiento de excepciones en el desarrollo de videojuegos, las excepciones se deberían utilizar para modelar situaciones realmente excepcionales [27], es decir, situaciones que nunca ocurren o que ocurren con poquísima frecuencia. Cuando se lanza una excepción se pone en marcha un proceso complejo compuesto de diversas operaciones, como por ejemplo la búsqueda del bloque de manejo de excepciones más cercano, la modificación de la pila, la destrucción de objetos y la transferencia del control del programa al bloque *catch*.

Este proceso es lento y afectaría enormemente al **rendimiento** del programa. Sin embargo, en el ámbito del desarrollo de videojuegos esta situación no resulta tan crítica, debido a que el uso de excepciones se limita, generalmente, a casos realmente excepcionales. En otras palabras, las excepciones no se utilizan para detectar que, por ejemplo, un enemigo ha caído al agua, sino para modelar aspectos críticos como que el sistema se esté quedando sin memoria física. Al final, este tipo de situaciones puede conducir a una eventual parada del sistema, por lo que el impacto de una excepción no resulta tan importante.

En este contexto, las **consolas de videojuegos** representan el caso más extremo, ya que normalmente tienen los recursos acotados y hay que ser especialmente cuidadoso con el rendimiento de los juegos. El caso más representativo es el **tamaño de la memoria principal**, en el que el impacto de utilizar excepciones puede ser desastroso. Sin embargo, en un PC este problema no es tan crítico, por lo que el uso de excepciones puede ser más recomendable. Algunos autores recomiendan no hacer uso de excepciones [42], sino de códigos de error, en el desarrollo para consolas de videojuegos, debido a su limitada capacidad de memoria.

También es importante reflexionar sobre el **impacto** del uso de las excepciones cuando no se lanzan, es decir, debido principalmente a la inclusión de sentencias *try-catch*. En general, este impacto está vinculado al compilador y a la forma que éste tiene para tratar con esta situación. Otro aspecto relevante es dónde se usa este tipo de sentencias. Si es en fases del juego como la carga inicial de recursos, por ejemplo mapas o modelos, se puede asumir perfectamente la degradación del rendimiento. Por el contrario, si dichas sentencias se van a ejecutar en un módulo que se ejecuta continuamente, entonces el rendimiento se verá afectado enormemente.

En general, las excepciones deberían usarse en aquellas partes de código en las que es posible que ocurra un error de manera inesperada. En el caso particular de los videojuegos, algunos ejemplos representativos son la detección de un archivo corrupto, un fallo hardware o una desconexión de la red.

El uso de excepciones puede coexistir perfectamente con el uso de valores de retorno, aunque es importante tener claro cuándo utilizar un planteamiento y cuándo utilizar otro. Por ejemplo, en el ámbito del desarrollo de videojuegos, el uso de valores de retorno es una solución más práctica y limpia si lo que se desea es simplemente conocer si una función terminó su ejecución de manera adecuada o no, independientemente de los tipos de errores que puedan producirse.

Capítulo 4

Patrones de Diseño

Cleto Martín Angelina
Francisco Moya Fernández

Cuando nos enfrentamos al diseño de un programa informático como un videojuego, no es posible abordarlo por completo. El proceso de diseño de una aplicación suele ser *iterativo* y en diferentes etapas de forma que se vaya *refinando* con el tiempo. El diseño perfecto y a la primera es muy difícil de conseguir.

La tarea de diseñar aplicaciones es compleja y, con seguridad, una de las más importantes y que más impacto tiene no sólo sobre el producto final, sino también sobre su vida futura. En el diseño de la aplicación es donde se definen las *estructuras* y *entidades* que se van a encargar de resolver el problema modelado, así como sus *relaciones* y sus *dependencias*. Cómo de bien definamos estas entidades y relaciones influirá, en gran medida, en el éxito o fracaso del proyecto y en la viabilidad de su mantenimiento.

El diseño, por tanto, es una tarea capital en el ciclo de vida del software. Sin embargo, no existe un procedimiento sistemático y claro sobre cómo crear el mejor diseño para un problema dado. Podemos utilizar metodologías, técnicas y herramientas que nos permitan refinar nuestro diseño. La *experiencia* también juega un papel importante. Sin embargo, el contexto de la aplicación es crucial y los requisitos, que pueden cambiar durante el proceso de desarrollo, también.

Un videojuego es un programa con una componente muy creativa. Además, el mercado de videojuegos se mueve muy deprisa y la adaptación a ese medio "cambiante" es un factor determinante. En general, los diseños de los programas deben ser *escalables*, *extensibles* y que permitan crear componentes *reutilizables*.

Esta última característica es muy importante ya que la experiencia nos hace ver que al construir una aplicación se nos presentan situaciones *recurrentes* y que se asemejan a situaciones pasadas. Es el *deja vú* en el diseño: «¿cómo solucioné esto?». En esencia, muchos componentes pueden modelarse de formas similares.

En este capítulo, se describen algunos *patrones de diseño* que almacenan este conocimiento experimental de diseño procedente del estudio de aplicaciones y de los éxitos y fracasos de casos reales. Bien utilizados, permiten obtener un mejor diseño más temprano.

4.1. Introducción

El diseño de una aplicación es un proceso iterativo y de continuo refinamiento. Normalmente, una aplicación es lo suficientemente compleja como para que su diseño tenga que ser realizado por etapas, de forma que al principio se identifican los módulos más abstractos y, progresivamente, se concreta cada módulo con un diseño en particular.

En el camino, es común encontrar problemas y situaciones que conceptualmente pueden parecerse entre sí, por lo menos a priori. Quizás un estudio más exhaustivo de los requisitos permitan determinar si realmente se trata de problemas equivalentes.

Por ejemplo, supongamos que para resolver un determinado problema se llega a la conclusión de que varios tipos de objetos deben esperar a un evento producido por otro. Esta situación puede darse en la creación de una interfaz gráfica donde la pulsación de un botón dispara la ejecución de otras acciones. Pero también es similar a la implementación de un manejador del teclado, cuyas pulsaciones son recogidas por los procesos interesados, o la de un gestor de colisiones, que notifica choques entre elementos del juego. Incluso se parece a la forma en que muchos programas de chat envían mensajes a un grupo de usuarios.

Ciertamente, cada uno de los ejemplos anteriores tiene su contexto y no es posible (ni a veces deseable) aplicar exactamente la misma solución a cada uno de ellos. Sin embargo, sí que es cierto que existe semejanza en la *esencia* del problema. En nuestro ejemplo, en ambos casos existen entidades que necesitan ser notificadas cuando ocurre un cierto evento.

Definición

En [38], un patrón de diseño es *una descripción de la comunicación entre objetos y clases personalizadas para solucionar un problema genérico de diseño bajo un contexto determinado.*

Los patrones de diseño son formas bien **conocidas** y **probadas** de resolver problemas de diseño que son recurrentes en el tiempo. Los patrones de diseño son ampliamente utilizados en las disciplinas creativas y técnicas. Así, de la misma forma que un guionista de cine crea guiones a partir de patrones argumentales como «comedia» o «ciencia-ficción», un ingeniero se basa en la experiencia de otros proyectos para identificar patrones comunes que le ayuden a diseñar nuevos procesos. De esta forma, **reutilizando** soluciones bien probadas y conocidas se ayuda a reducir el tiempo necesario para el diseño.

Los patrones sintetizan la tradición y experiencia profesional de diseñadores de software experimentados que han evaluado y demostrado que la solución proporcionada es una **buena solución** bajo un determinado contexto. El diseñador o desarrollador que conozca diferentes patrones de diseño podrá reutilizar estas soluciones, pudiendo alcanzar un mejor diseño más rápidamente.

4.1.1. Estructura de un patrón de diseño

Cuando se describe un patrón de diseño se pueden citar más o menos propiedades del mismo: el problema que resuelve, sus ventajas, si proporciona escalabilidad en el diseño o no, etc. Nosotros vamos a seguir las directrices marcadas por los autores del famoso libro de Design Patterns [38] [1], por lo que para definir un patrón de diseño es necesario describir, como mínimo, cuatro componentes fundamentales:

- **Nombre**: el nombre del patrón es fundamental. Es deseable tener un nombre corto y autodefinido, de forma que sea fácil de manejar por diseñadores y desarrolladores.

  Los buenos nombres pueden ser compartidos por todos de forma que se cree un *vocabulario* común con el que se pueda describir documentación fácilmente, además de construir y detallar soluciones más complejas basadas en patrones. También se deben indicar los *alias* del patrón.

- **Problema y contexto**: obviamente, el problema que resuelve un patrón en concreto debe ser descrito detalladamente. Sin embargo, es muy importante que se dé una definición clara del contexto en el que el patrón tiene sentido aplicarlo. El contexto se puede ver como un listado de precondiciones que deben ser cumplidas para poder aplicar el patrón.

- **Solución**: la solución que proporciona un patrón se describe genéricamente y nunca ligada a ninguna implementación. Normalmente, se utiliza los conceptos y nomenclatura de la programación orientada objetos. Por ello, la solución normalmente describe las clases y las relaciones entre objetos, así como la responsabilidad de cada entidad y cómo colaboran entre ellas para llegar a la solución.

  Gracias a la adopción de esta nomenclatura, la implementación de los patrones en los lenguajes orientados a objetos como C++ es más directa dada su especificación abstracta.

- **Ventajas y desventajas**: la aplicación de un patrón de diseño no es una decisión que debe tomarse sin tener en cuenta los beneficios que aporta y sus posibles inconvenientes. Junto con los anteriores apartados, se deben especificar las ventajas y desventajas que supone la aplicación del patrón en diferentes términos: complejidad, tiempo de ejecución, acoplamiento, cohesión, extensibilidad, portabilidad, etc. Si estos términos están documentados, será más sencillo tomar una decisión.

El uso de patrones de diseño es *recomendable*. Sin embargo, hay que tener en cuenta que un patrón no es bueno ni malo en sí mismo ni en su totalidad. El **contexto de aplicación** puede ser determinante para no optar por una solución basada en un determinado patrón. Los cañones pueden ser una buena arma de guerra, pero no para matar moscas.

[1] También conocido como *The Gang of Four (GoF)* (la «Banda de los Cuatro») en referencia a los autores del mismo. Sin duda se trata de un famoso libro en este área, al que no le faltan detractores.

4.1.2. Tipos de patrones

De entre los diferentes criterios que se podrían adoptar para clasificar los diferentes patrones de diseño, uno de los más aceptados es el ámbito de diseño donde tienen aplicación. De esta forma, se definen tres categorías fundamentales:

- Patrones **de creación**: se trata de aquellos que proporcionan una solución relacionada con la construcción de clases, objetos y otras estructuras de datos. Por ejemplo, patrones como *Abstract Factory*, *Builder* y otros ofrecen mecanismos de creación de instancias de objetos y estructuras escalables dependiendo de las necesidades.
- Patrones **estructurales**: este tipo de patrones versan sobre la forma de organizar las jerarquías de clases, las relaciones y las diferentes composiciones entre objetos para obtener un buen diseño en base a unos requisitos de entrada. Patrones como *Adapter*, *Facade* o *Flyweight* son ejemplos de patrones estructurales.
- Patrones **de comportamiento**: las soluciones de diseño que proporcionan los patrones de comportamiento están orientadas al envío de mensajes entre objetos y cómo organizar ejecuciones de diferentes métodos para conseguir realizar algún tipo de tarea de forma más conveniente. Algunos ejemplos son *Visitor*, *Iterator* y *Observer*.

Algunos profesionales, como Jeff Atwood, critican el uso «excesivo» de patrones. Argumentan que es más importante identificar bien las responsabilidades de cada entidad que el propio uso del patrón. Otros se plantean si el problema no está realmente en los propios lenguajes de programación, que no proporcionan las herramientas semánticas necesarias.

4.2. Patrones de creación

En esta sección se describen algunos patrones que ayudan en el diseño de problemas en los que la creación de instancias de diferentes tipos es el principal problema.

4.2.1. Singleton

El patrón *singleton* se suele utilizar cuando se requiere tener una **única instancia** de un determinado tipo de objeto.

La idoneidad del patrón Singleton es muy controvertida y está muy cuestionada. Muchos autores y desarrolladores, entre los que destaca Eric Gamma (uno de los autores de [38]) consideran que es un antipatrón, es decir, una mala solución a un problema de diseño.)

Problema

En C++, utilizando el operador `new` es posible crear una instancia de un objeto. Sin embargo, es posible que necesitemos que sólo exista una instancia de una clase determinada por diferentes motivos (prevención de errores, seguridad, etc.).

El balón en un juego de fútbol o la entidad que representa al mundo 3D son ejemplos donde podría ser conveniente mantener una única instancia de este tipo de objetos.

Solución

Para garantizar que sólo existe una instancia de una clase es necesario que los clientes no puedan acceder directamente al constructor. Por ello, en un singleton el constructor es, por lo menos, `protected`. A cambio se debe proporcionar un único punto (controlado) por el cual se pide la instancia única. El diagrama de clases de este patrón se muestra en la figura 4.1.

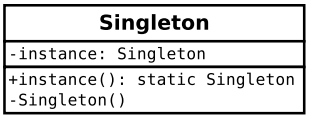


Figura 4.1: Diagrama de clases del patrón Singleton.

Implementación

A continuación, se muestra una implementación básica del patrón Singleton:

Listado 4.1: Singleton (ejemplo)

```
1  /* Header */
2  class Ball {
3  protected:
4
5    float _x, _y;
6    static Ball* theBall_;
7
8    Ball(float x, float y) : _x(x), _y(y) { };
9    Ball(const Ball& ball);
10   void operator=(const Ball& ball ) ;
11
12 public:
13   static Ball& getTheBall();
14
15   void move(float _x, float _y) { /*...*/ };
16 };
17
18 Ball& Ball::getTheBall()
19 {
20   static Ball theBall_;
21   return theBall_;
22 }
```

Como se puede ver, la característica más importante es que los métodos que pueden crear una instancia de `Ball` son todos privados para los clientes externos. Todos ellos deben utilizar el método estático `getTheBall()` para obtener la única instancia.

Esta implementación `no` es válida para programas multihilo, es decir, no es *thread-safe*.

Como ejercicio se plantea la siguiente pregunta: en la implementación proporcionada, ¿se garantiza que no hay *memory leak*? ¿Qué sería necesario para que la implementación fuera *thread-safe*?

Consideraciones

El patrón Singleton puede ser utilizado para modelar:

- Gestores de acceso a base de datos, sistemas de ficheros, render de gráficos, etc.
- Estructuras que representan la configuración del programa para que sea accesible por todos los elementos en cualquier instante.

El Singleton es un caso particular de un patrón de diseño más general llamado *Object Pool*, que permite crear n instancias de objetos de forma controlada.

4.2.2. Abstract Factory

El patrón *Abstract Factory* permite crear diferentes tipos de instancias, aislando al cliente sobre cómo se debe crear cada una de ellas.

Problema

Conforme un programa crece, el número de clases que representan los diferentes tipos de objetos suele también crecer. Muchos de los diseños tienen jerarquías de objetos tales como la que se muestra en la figura 4.2.

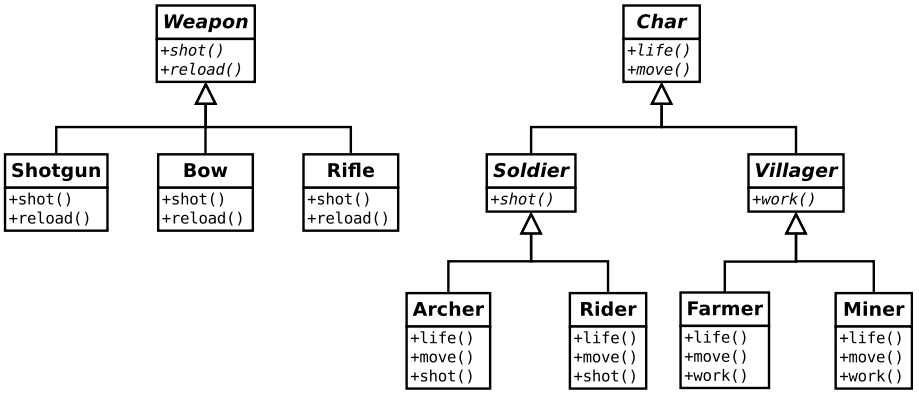


Figura 4.2: Ejemplos de jerarquías de clases

En ella, se muestra jerarquías de clases que modelan los diferentes tipos de personajes de un juego y algunas de sus armas. Para construir cada tipo de personaje es necesario saber cómo construirlo y con qué otro tipo de objetos tiene relación. Por ejemplo, restricciones del tipo «la gente del pueblo no puede llevar armas» o «los arqueros sólo pueden puede tener un arco», es conocimiento específico de la clase que se está construyendo.

Supongamos que en nuestro juego, queremos obtener razas de personajes: hombres y orcos. Cada raza tiene una serie de características propias que hacen que pueda moverse más rápido, trabajar más o tener más resistencia a los ataques.

El patrón Abstract Factory puede ser de ayuda en este tipo de situaciones en las que es necesario crear diferentes tipos de objetos utilizando una jerarquía de componentes. Dada la complejidad que puede llegar a tener la creación de una instancia es deseable aislar la forma en que se construye cada clase de objeto.

Solución

En la figura 4.3 se muestra la aplicación del patrón para crear las diferentes razas de soldados. Por simplicidad, sólo se ha aplicado a esta parte de la jerarquía de personajes.

En primer lugar se define una *factoría abstracta* que será la que utilice el cliente (`Game`) para crear los diferentes objetos. `CharFactory` es una factoría que sólo define métodos abstractos y que serán implementados por sus clases hijas. Éstas son factorías concretas a cada tipo de raza (`ManFactory` y `OrcFactory`) y ellas son las que crean las instancias concretas de objetos `Archer` y `Rider` para cada una de las razas.

En definitiva, el patrón Abstract Factory recomienda crear las siguientes entidades:

- Factoría abstracta que defina una interfaz para que los clientes puedan crear los distintos tipos de objetos.
- Factorías concretas que realmente crean las instancias finales.

Implementación

Basándonos en el ejemplo anterior, el objeto `Game` sería el encargado de crear los diferentes personajes utilizando una factoría abstracta. El siguiente fragmento de código muestra cómo la clase `Game` recibe una factoría concreta (utilizando polimorfismo) y la implementación del método que crea los soldados.

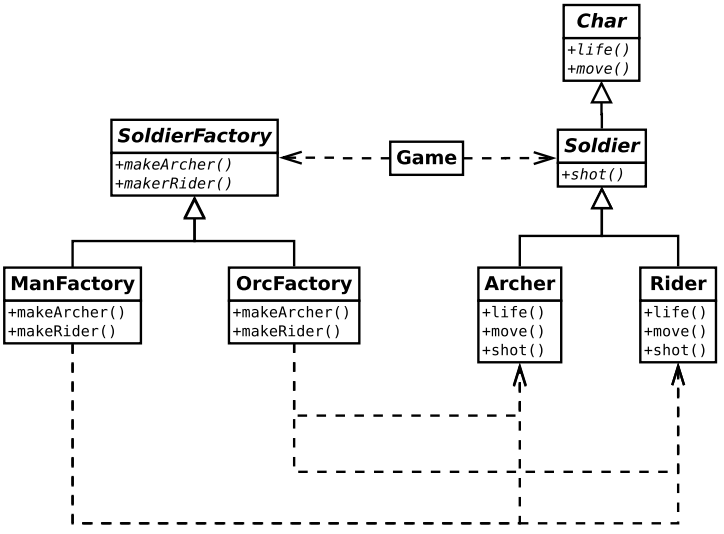


Figura 4.3: Aplicación del patrón Abstract Factory

Listado 4.2: Abstract Factory (Game)

```
/* ... */
Game game;
SoldierFactory* factory;

if (isSelectedMan) {
  factory = new ManFactory();
 } else {
  factory = new OrcFactory();
 }

game->createSoldiers(factory);
/* ... */


/* Game implementation */
vector<Soldier*> Game::createSoldiers(SoldierFactory* factory)
{
  vector<Solider*> soldiers;
  for (int i=0; i<5; i++) {
    soldiers.push_back(factory->makeArcher());
    soldiers.push_back(factory->makeRider());
  }
  return soldiers;
}
```

Como puede observarse, la clase `Game` simplemente invoca los métodos de la factoría abstracta. Por ello, `createSoldier()` funciona exactamente igual para cualquier tipo de factoría concreta (de hombres o de orcos).

Una implementación del método `makeArcher()` de cada factoría concreta podría ser como sigue:

Listado 4.3: Abstract Factory (factorías concretas)

```
/* OrcFactory */
Archer* OrcFactory::makeArcher()
{
  Archer archer = new Archer();
  archer->setLife(200);
  archer->setName('Orc');
  return archer;
}

/* ManFactory */
Archer* ManFactory::makeArcher()
{
  Archer archer = new Archer();
  archer->setLife(100);
  archer->setName('Man');
  return archer;
}
```

Nótese como las factorías concretas ocultan las particularidades de cada tipo. Una implementación similar tendría el método `makeRider()`.

Consideraciones

El patrón Abstract Factory puede ser aplicable cuando:

- el sistema de creación de instancias debe aislarse.

- es necesaria la creación de varias instancias de objetos para tener el sistema configurado.
- cuando la creación de las instancias implican la imposición de restricciones y otras particularidades propias de los objetos que se construyen.
- los productos que se deben fabricar en las factorías no cambian excesivamente en el tiempo. Añadir nuevos productos implica añadir métodos a todas las factorías ya creadas, por lo que es un poco problemático. En nuestro ejemplo, si quisiéramos añadir un nuevo tipo de soldado deberíamos modificar la factoría abstracta y las concretas. Por ello, es recomendable que se aplique este patrón sobre diseños con un cierto grado de estabilidad.

Un patrón muy similar a éste es el patrón *Builder*. Con una estructura similar, el patrón Builder se centra en el proceso de *cómo* se crean las instancias y no en la jerarquía de factorías que lo hacen posible. Como ejercicio se plantea estudiar el patrón Builder y encontrar las diferencias.

4.2.3. Factory Method

El patrón *Factory Method* se basa en la definición de una interfaz para crear instancias de objetos y permite a las subclases decidir cómo se crean dichas instancias implementando un método determinado.

Problema

Al igual que ocurre con el patrón Abstract Factory, el problema que se pretende resolver es la creación de diferentes instancias de objetos abstrayendo la forma en que realmente se crean.

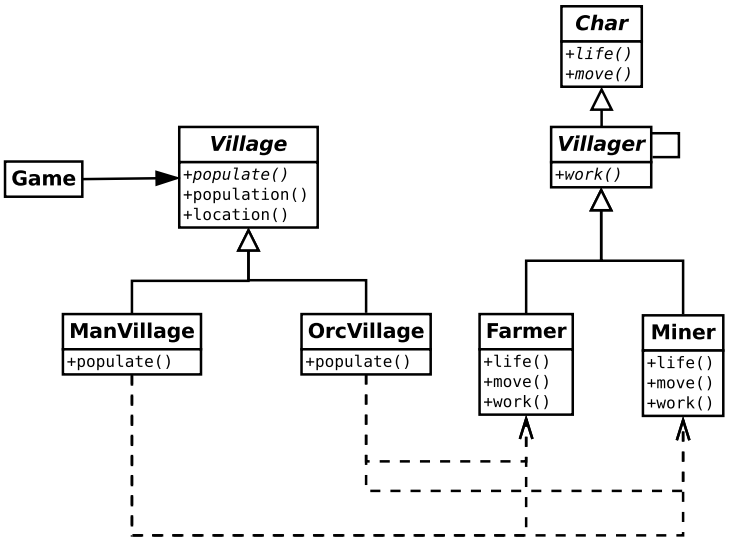


Figura 4.4: Ejemplo de aplicación de Factory Mehod

Solución

La figura 4.4 muestra un diagrama de clases para nuestro ejemplo que emplea el patrón Factory Method para crear ciudades en las que habitan personajes de diferentes razas.

Como puede verse, los objetos de tipo `Village` tienen un método `populate()` que es implementado por las subclases. Este método es el que crea las instancias de `Villager` correspondientes a cada raza. Este método es el *método factoría*. Además de este método, también se proporcionan otros como `population()` que devuelve la población total, o `location()` que devuelve la posición de la cuidad en el mapa. Todos estos métodos son comunes y heredados por las ciudades de hombres y orcos.

Finalmente, objetos `Game` podrían crear ciudades y, consecuentemente, crear ciudadanos de distintos tipos de una forma transparente.

Consideraciones

Este patrón presenta las siguientes características:

- No es necesario tener una factoría o una jerarquía de factorías para la creación de objetos. Permite diseños más adaptados a la realidad.
- El método factoría, al estar integrado en una clase, hace posible conectar dos jerarquía de objetos distintas. Por ejemplo, si los personajes tienen un método factoría de las armas que pueden utilizar, el dominio de las armas y los personajes queda unido a través el método. Las subclases de los personajes crearían las instancias de `Weapon` correspondientes.

Nótese que el patrón Factory Method se utiliza para implementar el patrón Abstract Factory ya que la factoría abstracta define una interfaz con métodos de construcción de objetos que son implementados por las subclases.

4.2.4. Prototype

El patrón *Prototype* proporciona abstracción a la hora de crear diferentes objetos en un contexto donde se desconoce cuántos y cuáles deben ser creados a priori. La idea principal es que los objetos deben poder clonarse en tiempo de ejecución.

Problema

Los patrones Factory Method y Abstract Factory tienen el problema de que se basan en la herencia e implementación de métodos abstractos por subclases para definir cómo se construye cada producto concreto. Para sistemas donde el número de productos concretos puede ser elevado o indeterminado esto puede ser un problema.

Supongamos que en nuestra jerarquía de armas, cuya clase padre es `Weapon`, comienza a crecer con nuevos tipos de armas y, además, pensamos en dejar libertad para que se carguen en tiempo de ejecución nuevas armas que se implementarán como librerías dinámicas. Además, el número de armas variará dependiendo de ciertas condiciones del juego y de configuración. En este contexto, puede ser más que dudoso el uso de factorías.

Solución

Para atender a las nuevas necesidades dinámicas en la creación de los distintos tipo de armas, sin perder la abstracción sobre la creación misma, se puede utilizar el patrón Prototype como se muestra en la figura 4.5.

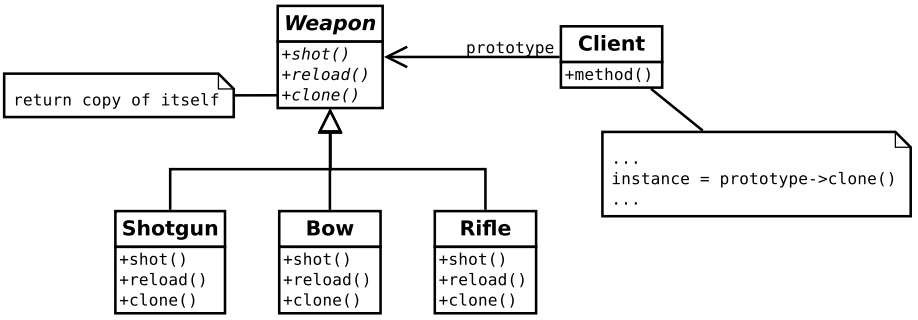


Figura 4.5: Ejemplo de aplicación de Prototype

La diferencia fundamental se encuentra en la adición del método `clone()` a todas los productos que pueden ser creados. El cliente del prototipo sólo tiene que invocar `clone()` sobre su instancia `Weapon` para que se cree una instancia concreta. Como se puede ver, no es necesario un agente intermedio (factorías) para crear instancias de un determinado tipo. La creación se realiza en la clase concreta que representa a la instancia, por lo que basta con cambiar la instancia `prototype` de `Client` para que se creen nuevos tipos de objetos en tiempo de ejecución.

Consideraciones

Algunas notas interesantes sobre Prototype:

- Puede parecer que entra en conflicto con Abstract Factory debido a que intenta eliminar, precisamente, factorías intermedias. Sin embargo, es posible utilizar ambas aproximaciones en una *Prototype Abstract Factory* de forma que la factoría se configura con los prototipos concretos que puede crear y ésta sólo invoca a `clone()`.
- También es posible utilizar un *gestor de prototipos* que permita cargar y descargar los prototipos disponibles en tiempo de ejecución. Este gestor es interesante para tener diseños ampliables en tiempo de ejecución (*plugins*).
- Para que los objetos puedan devolver una copia de sí mismo es necesario que en su implementación esté el constructor de copia (*copy constructor*) que en C++ viene por defecto implementado.

4.3. Patrones estructurales

Hasta ahora, hemos visto patrones para diseñar aplicaciones donde el problema principal es la creación de diferentes instancias de clases. En esta sección se mostrarán los patrones de diseño estructurales que se centran en las relaciones entre clases y en cómo organizarlas para obtener un diseño eficiente para resolver un determinado problema.

4.3.1. Composite

El patrón *Composite* se utiliza para crear una organización arbórea y homogénea de instancias de objetos.

Problema

Para ilustrar el problema supóngase un juego de estrategia en el que los jugadores pueden recoger objetos o *items*, los cuales tienen una serie de propiedades como «precio», «descripción», etc. Cada item, a su vez, puede contener otros items. Por ejemplo, un bolso de cuero puede contener una pequeña caja de madera que, a su vez, contiene un pequeño reloj dorado.

En definitiva, el patrón Composite habla sobre cómo diseñar este tipo de estructuras recursivas donde la composición homogénea de objetos recuerda a una estructura arbórea.

Solución

Para el ejemplo expuesto anteriormente, la aplicación del patrón Composite quedaría como se muestran en la figura 4.6. Como se puede ver, todos los elementos son `Items` que implementan una serie de métodos comunes. En la jerarquía, existen *objetos compuestos*, como `Bag`, que mantienen una lista (`items`) donde residen los objetos que contiene. Naturalmente, los objetos compuestos suelen ofrecer también operaciones para añadir, eliminar y actualizar.

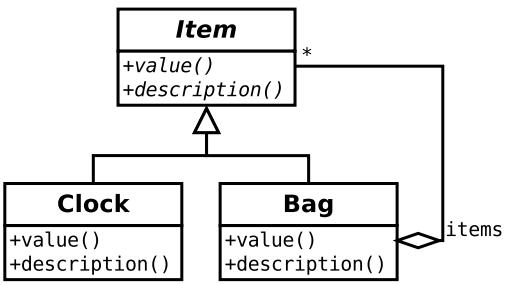


Figura 4.6: Ejemplo de aplicación del patrón Composite

Por otro lado, hay *objetos hoja* que no contienen a más objetos, como es el caso de `Clock`.

Consideraciones

Al utilizar este patrón, se debe tener en cuenta las siguientes consideraciones:

- Una buena estrategia para identificar la situación en la que aplicar este patrón es cuando tengo «un X y tiene varios objetos X».

- La estructura generada es muy flexible siempre y cuando no importe el tipo de objetos que pueden tener los objetos compuestos. Es posible que sea deseable prohibir la composición de un tipo de objeto con otro. Por ejemplo, un jarrón grande dentro de una pequeña bolsa. La comprobación debe hacerse en tiempo de ejecución y no es posible utilizar el sistema de tipos del compilador. En este sentido, usando Composite se relajan las restricciones de composición entre objetos.
- Los usuarios de la jerarquía se hacen más sencillos, ya que sólo tratan con un tipo abstracto de objeto, dándole homogeneidad a la forma en que se maneja la estructura.

4.3.2. Decorator

También conocido como *Wrapper*, el patrón *Decorator* sirve para añadir y/o modificar la responsabilidad, funcionalidad o propiedades de un objeto en tiempo de ejecución.

Problema

Supongamos que el personaje de nuestro videojuego porta un arma que utiliza para eliminar a sus enemigos. Dicha arma, por ser de un tipo determinado, tiene una serie de propiedades como el radio de acción, nivel de ruido, número de balas que puede almacenar, etc. Sin embargo, es posible que el personaje incorpore elementos al arma que puedan cambiar estas propiedades como un silenciador o un cargador extra.

El patrón Decorator permite organizar el diseño de forma que la incorporación de nueva funcionalidad en tiempo de ejecución a un objeto sea transparente desde el punto de vista del usuario de la clase *decorada*.

Solución

En la figura 4.7 se muestra la aplicación del patrón Decorator al supuesto anteriormente descrito. Básicamente, los diferentes tipos de armas de fuego implementan una clase abstracta llamada `Firearm`. Una de sus hijas es `FirearmDecorator` que es el padre de todos los componentes que «decoran» a un objeto `Firearm`. Nótese que este decorador implementa la interfaz propuesta por `Firearm` y está compuesta por un objeto `gun`, el cual decora.

Implementación

A continuación, se expone una implementación en C++ del ejemplo del patrón Decorator. En el ejemplo, un arma de tipo `Rifle` es decorada para tener tanto silenciador como una nueva carga de munición. Nótese cómo se utiliza la instancia `gun` a lo largo de los constructores de cada decorador.

Listado 4.4: Decorator

```
1 class Firearm {
2 public:
3   virtual float noise() const = 0;
4   virtual int bullets() const = 0;
5 };
6
7 class Rifle : public Firearm {
```

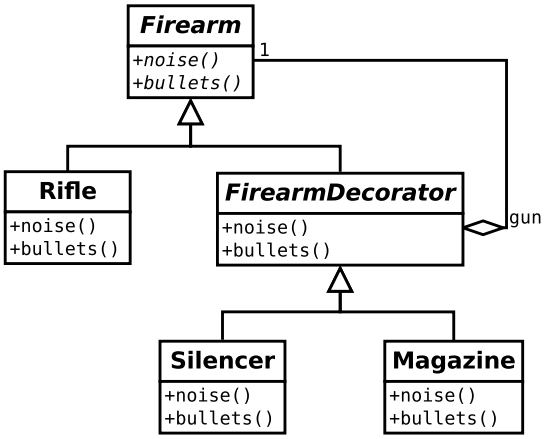


Figura 4.7: Ejemplo de aplicación del patrón Decorator

```
 8 public:
 9   float noise () const { return 150.0; }
10   int bullets () const { return 5; }
11 };
12
13 /* Decorators */
14
15 class FirearmDecorator : public Firearm {
16 protected:
17   Firearm* _gun;
18 public:
19   FirearmDecorator(Firearm* gun): _gun(gun) {};
20   virtual float noise () const { return _gun->noise(); }
21   virtual int bullets () const { return _gun->bullets(); }
22 };
23
24 class Silencer : public FirearmDecorator {
25 public:
26   Silencer(Firearm* gun) : FirearmDecorator(gun) {};
27   float noise () const { return _gun->noise() - 55; }
28   int bullets () const { return _gun->bullets(); }
29 };
30
31 class Magazine : public FirearmDecorator {
32 public:
33   Magazine(Firearm* gun) : FirearmDecorator(gun) {};
34   float noise () const { return _gun->noise(); }
35   int bullets () const { return _gun->bullets() + 5; }
36 };
37
38 /* Using decorators */
39
40 ...
41 Firearm* gun = new Rifle();
42 cout << "Noise: " << gun->noise() << endl;
43 cout << "Bullets: " << gun->bullets() << endl;
44 ...
45 // char gets a silencer
46 gun = new Silencer(gun);
47 cout << "Noise: " << gun->noise() << endl;
48 cout << "Bullets: " << gun->bullets() << endl;
49 ...
50 // char gets a new magazine
51 gun = new Magazine(gun);
52 cout << "Noise: " << gun->noise() << endl;
53 cout << "Bullets: " << gun->bullets() << endl;
```

En cada momento, ¿qué valores se imprimen?. Supón que el personaje puede quitar el silenciador. ¿Qué cambios habría que hacer en el código para «quitar» el decorador a la instancia?

Consideraciones

A la hora de aplicar el patrón Decorator se deben tener en cuenta las siguientes consideraciones:

- Es un patrón similar al Composite. Sin embargo, existen grandes diferencias:
  - Está más centrado en la **extensión de la funcionalidad** que en la composición de objetos para la generación de una jerarquía como ocurre en el Composite.
  - Normalmente, sólo existe un **objeto decorado** y no un vector de objetos (aunque también es posible).
- Este patrón permite tener una jerarquía de clases compuestas, formando una estructura más dinámica y flexible que la herencia estática. El diseño equivalente utilizando mecanismos de herencia debería considerar todos los posibles casos en las clases hijas. En nuestro ejemplo, habría 4 clases: rifle, rifle con silenciador, rifle con cargador extra y rifle con silenciador y cargador. Sin duda, este esquema es muy poco flexible.

4.3.3. Facade

El patrón *Facade* eleva el nivel de abstracción de un determinado sistema para ocultar ciertos detalles de implementación y hacer más sencillo su uso.

Problema

Muchos de los sistemas que proporcionan la capacidad de escribir texto en pantalla son complejos de utilizar. Su complejidad reside en su naturaleza generalista, es decir, están diseñados para abarcar un gran número de tipos de aplicaciones. Por ello, el usuario normalmente debe considerar cuestiones de «bajo nivel» como es configurar el propio sistema interconectando diferentes objetos entre sí que, a priori, parece que nada tienen que ver con la tarea que se tiene que realizar.

Para ver el problema que supone para un usuario un bajo nivel de abstracción no es necesario recurrir a una librería o sistema externo. Nuestro propio proyecto, si está bien diseñado, estará dividido en subsistemas que proporcionan una cierta funcionalidad. Basta con que sean genéricos y reutilizables para que su complejidad aumente considerablemente y, por ello, su uso sea cada vez más tedioso.

Por ejemplo, supongamos que hemos creado diferentes sistemas para realizar distintas operaciones gráficas (manejador de archivos, cargador de imágenes, etc.) el siguiente código correspondería con la animación de una explosión en un punto determinado de la pantalla.

Sin duda alguna, y pese a que ya se tienen objetos que abstraen subsistemas tales como sistemas de archivos, para los clientes que únicamente quieran mostrar explosiones no proporciona un nivel de abstracción suficiente. Si esta operación se realiza frecuentemente el problema se agrava.

Listado 4.5: Ejemplo de uso de diferentes subsistemas

```
File* file_exp1 = FileManager::load_file("explosion1.tif");
File* file_exp2 =  FileManager::load_file("explosion2.tif");
Image* explosion1 = ImageManager::get_image_from_file(file_exp1);
Image* explosion2 = ImageManager::get_image_from_file(file_exp2);
Screen* screen = Screen::get_screen();

screen->add_element(explosion1, x, y);
screen->add_element(explosion2, x+2, y+2);
...
/* more configuration */
...
screen->draw();
```

Solución

Utilizando el patrón Facade, se proporciona un mayor nivel de abstracción al cliente de forma que se construye una clase «fachada» entre él y los subsistemas con menos nivel de abstracción. De esta forma, se proporciona una visión unificada del conjunto y, además, se controla el uso de cada componente.

Para el ejemplo anterior, se podría crear una clase que proporcione la una funcionalidad más abstracta. Por ejemplo, algo parecido a lo siguiente::

Listado 4.6: Simplificación utilizando Facade

```
AnimationManager* animation = new AnimationManager();
animation->explosion_at(3,4);
```

Como se puede ver, el usuario ya no tiene que conocer las relaciones que existen entre los diferentes módulos para crear este tipo de animaciones. Esto aumenta, levemente, el nivel de abstracción y hace más sencillo su uso.

En definitiva, el uso del patrón Facade proporciona una estructura de diseño como la mostrada en la figura 4.8.

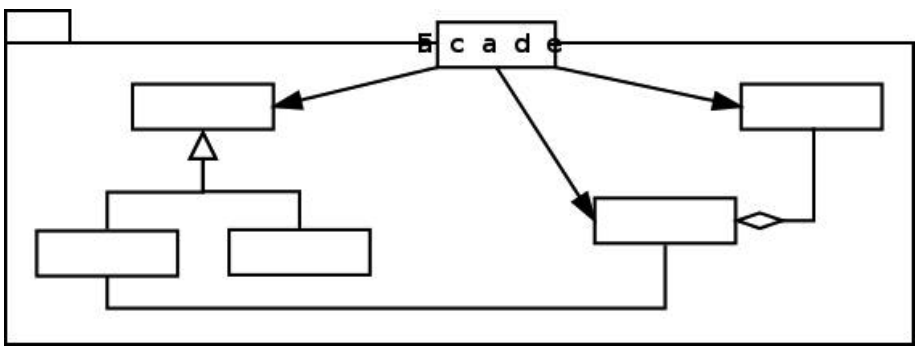


Figura 4.8: Ejemplo de aplicación del patrón Facade

Consideraciones

El patrón Facade puede ser útil cuando:

- Es necesario *refactorizar*, es decir, extraer funcionalidad común de los sistemas y agruparla en función de las necesidades.
- Los sistemas deben ser independientes y portables.
- Controlar el acceso y la forma en que se utiliza un sistema determinado.
- Los clientes pueden seguir utilizando los subsistemas directamente, sin pasar por la fachada, lo que da la flexibilidad de elegir entre una implementación de bajo nivel o no.

Sin embargo, utilizando el patrón Facade es posible caer en los siguientes errores:

- Crear clases con un tamaño desproporcionado. Las clases fachada pueden contener demasiada funcionalidad si no se divide bien las responsabilidades y se tiene claro el objetivo para el cual se creo la fachada. Para evitarlo, es necesario ser crítico/a con el nivel de abstracción que se proporciona.
- Obtener diseños poco flexibles y con mucha contención. A veces, es posible crear fachadas que obliguen a los usuarios a un uso demasiado rígido de la funcionalidad que proporciona y que puede hacer que sea más cómodo, a la larga, utilizar los subsistemas directamente. Además, una fachada puede convertirse en un único punto de fallo, sobre todo en sistemas distribuidos en red.
- Exponer demasiados elementos y, en definitiva, no proporcionar un nivel de abstracción adecuado.

4.3.4. MVC

El patrón MVC (Model View Controller) se utiliza para aislar el dominio de aplicación, es decir, la lógica, de la parte de presentación (interfaz de usuario).

Problema

Programas como los videojuegos requieren la interacción de un usuario que, normalmente, realiza diferentes acciones sobre una interfaz gráfica. Las interfaces disponibles son muy variadas: desde aplicaciones de escritorio con un entorno GTK (GIMP ToolKit) a aplicaciones web, pasando por una interfaz en 3D creada para un juego determinado.

Supongamos que una aplicación debe soportar varios tipos de interfaz a la vez. Por ejemplo, un juego que puede ser utilizado con una aplicación de escritorio y, también, a través de una página web. El patrón MVC sirve para aislar la lógica de la aplicación, de la forma en que se ésta se presenta, su interfaz gráfica.

Solución

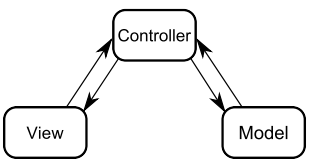


Figura 4.9: Estructura del patrón MVC.

En el patrón MVC, mostrado en la figura 4.9, existen tres entidades bien definidas:

- **Vista**: se trata de la interfaz de usuario que interactúa con el usuario y recibe sus órdenes (pulsar un botón, introducir texto, etc.). También recibe órdenes desde el controlador, para mostrar información o realizar un cambio en la interfaz.

- **Controlador**: el controlador recibe órdenes utilizando, habitualmente, manejadores o *callbacks* y traduce esa acción al dominio del modelo de la aplicación. La acción puede ser crear una nueva instancia de un objeto determinado, actualizar estados, pedir operaciones al modelo, etc.
- **Modelo**: el modelo de la aplicación recibe las acciones a realizar por el usuario, pero ya independientes del tipo de interfaz utilizado porque se utilizan, únicamente, estructuras propias del dominio del modelo y llamadas desde el controlador.

  Normalmente, la mayoría de las acciones que realiza el controlador sobre el modelo son operaciones de consulta de su estado para que pueda ser convenientemente representado por la vista.

  MVC no es patrón con una separación tan rígida. Es posible encontrar implementaciones en las que, por ejemplo, el modelo notifique directamente a las interfaces de forma asíncrona eventos producidos en sus estructuras y que deben ser representados en la vista (siempre y cuando exista una aceptable independencia entre las capas). Para ello, es de gran utilidad el patrón Observer (ver sección 4.4.1).

Consideraciones

El patrón MVC es la filosofía que se utiliza en un gran número de entornos de ventanas. Sin embargo, muchos sistemas web como Django también se basan en este patrón. Sin duda, la división del código en estos roles proporciona flexibilidad a la hora de crear diferentes tipos de presentaciones para un mismo dominio.

De hecho, desde un punto de vista general, la estructura más utilizada en los videojuegos se asemeja a un patrón MVC: la interfaz gráfica utilizando gráficos 3D/2D (vista), bucle de eventos (controlador) y las estructuras de datos internas (modelo).

4.3.5. Adapter

El patrón *Adapter* se utiliza para proporcionar una interfaz que, por un lado, cumpla con las demandas de los clientes y, por otra, haga compatible otra interfaz que, a priori, no lo es.

Problema

Es muy probable que conforme avanza la construcción de la aplicación, el diseño de las interfaces que ofrecen los componentes pueden no ser las adecuadas o, al menos, las esperadas por los usuarios de los mismos. Una solución rápida y directa es adaptar dichas interfaces a nuestras necesidades. Sin embargo, esto puede que no sea tan sencillo.

En primer lugar, es posible que no tengamos la posibilidad de modificar el código de la clase o sistema que pretendemos cambiar. Por otro lado, puede ser que sea un requisito no funcional por parte del cliente: determinado sistema o biblioteca debe utilizarse sí o sí. Si se trata de una biblioteca externa (*third party*), puede ocurrir que la modificación suponga un coste adicional para el proyecto ya que tendría que ser mantenida por el propio proyecto y adaptar las mejoras y cambios que se añadan en la versión no modificada.

Por lo tanto, es posible llegar a la conclusión de que a pesar de que el sistema, biblioteca o clase no se adapta perfectamente a nuestras necesidades, trae más a cuenta utilizarla que hacerse una versión propia.

Solución

Usando el patrón Adapter es posible crear una nueva interfaz de acceso a un determinado objeto, por lo que proporciona un mecanismo de *adaptación* entre las demandas del objeto cliente y el objeto servidor que proporciona la funcionalidad.

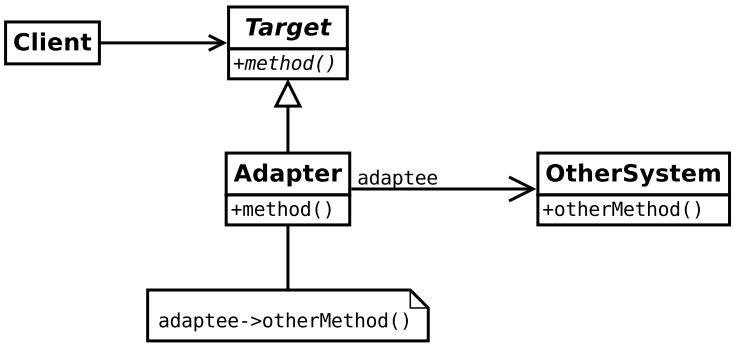


Figura 4.10: Diagrama de clases del patrón Adapter

En la figura 4.10 se muestra un diagrama de clases genérico del patrón basado en la composición. Como puede verse, el cliente no utiliza el sistema adaptado, sino el adaptador. Este es el que transforma la invocación a `method()` en `otherMethod()`. Es posible que el adaptador también incluya nueva funcionalidad. Algunas de las más comunes son:

- La comprobación de la corrección de los parámetros.
- La transformación de los parámetros para ser compatibles con el sistema adaptado.

Consideraciones

Algunas consideraciones sobre el uso del patrón Adapter:

- Tener sistemas muy reutilizables puede hacer que sus interfaces no puedan ser compatibles con una común. El patrón Adapter es una buena opción en este caso.
- Un mismo adaptador puede utilizarse con varios sistemas.
- Otra versión del patrón es que la clase Adapter sea una subclase del sistema adaptado. En este caso, la clase Adapter y la adaptada tienen una relación más estrecha que si se realiza por composición.
- Este patrón se parece mucho al Decorator. Sin embargo, difieren en que la finalidad de éste es proporcionar una interfaz completa del objeto adaptador, mientras que el decorador puede centrarse sólo en una parte.

4.3.6. Proxy

El patrón *Proxy* proporciona mecanismos de abstracción y control para acceder a un determinado objeto «simulando» que se trata del objeto real.

Problema

Muchos de los objetos de los que puede constar una aplicación pueden presentar diferentes problemas a la hora de ser utilizados por clientes:

- **Coste** computacional: es posible que un objeto, como una imagen, sea costoso de manipular y cargar.
- Acceso **remoto**: el acceso por red es una componente cada vez más común entre las aplicaciones actuales. Para acceder a servidores remotos, los clientes deben conocer las interioridades y pormenores de la red (*sockets*, protocolos, etc.).
- Acceso **seguro**: es posible que muchos objetos necesiten diferentes privilegios para poder ser utilizados. Por ejemplo, los clientes deben estar autorizados para poder acceder a ciertos métodos.
- Dobles de **prueba**: a la hora de diseñar y probar el código, puede ser útil utilizar objetos *dobles* que reemplacen instancias reales que pueden hacer que las pruebas sea pesadas y/o lentas.

Solución

Supongamos el problema de mostrar una imagen cuya carga es costosa en términos computacionales. La idea detrás del patrón Proxy (ver figura 4.11) es crear una un objeto intermedio (`ImageProxy`) que representa al objeto real (`Image`) y que se utiliza de la misma forma desde el punto de vista del cliente. De esta forma, el objeto proxy puede cargar una única vez la imagen y mostrarla tantas veces como el cliente lo solicite.

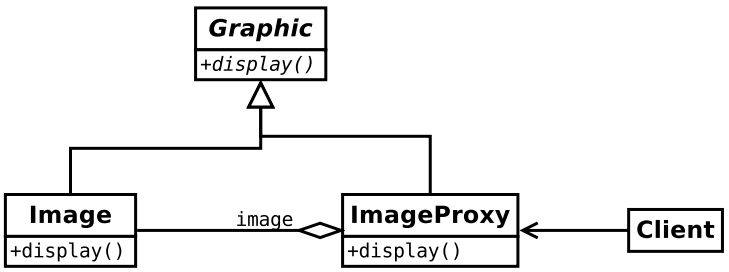


Figura 4.11: Ejemplo de aplicación del patrón Proxy

Implementación

A continuación, se muestra una implementación del problema anteriormente descrito donde se utiliza el patrón Proxy. En el ejemplo puede verse (en la parte del cliente) cómo la imagen sólo se carga una vez: la primera vez que se invoca a `display()`. El resto de invocaciones sólo muestran la imagen ya cargada.

Listado 4.7: Ejemplo de implementación de Proxy

```
1 class Graphic {
2 public:
3   void display() = 0;
4 };
5
```

```
6  class Image : public Graphic {
7  public:
8    void load() {
9      ...
10     /* perform a hard file load */
11     ...
12   }
13
14   void display() {
15     ...
16     /* perform display operation */
17     ...
18   }
19 };
20
21 class ImageProxy : public Graphic {
22 private:
23   Image* _image;
24 public:
25   void display() {
26     if (not _image) {
27       _image = new Image();
28       _image.load();
29     }
30     _image->display();
31   }
32 };
33
34 /* Client */
35 ...
36 Graphic image = new ImageProxy();
37 image->display(); // loading and display
38 image->display(); // just display
39 image->display(); // just display
40 ...
```

Consideraciones

Existen muchos ejemplos donde se hace un uso intensivo del patrón proxy en diferentes sistemas:

- En los sistemas de autenticación, dependiendo de las credenciales presentadas por el cliente, devuelven un proxy u otro que permiten realizar más o menos operaciones.
- En middlewares orientados a objetos como CORBA o ZeroC ICE (Internet Communication Engine), se utiliza la abstracción del Proxy para proporcionar invocaciones remotas entre objetos distribuidos. Desde el punto de vista del cliente, la invocación se produce como si el objeto estuviera accesible localmente. El proxy es el encargado de proporcionar esta abstracción.

4.4. Patrones de comportamiento

Los patrones de diseño relacionados con el comportamiento de las aplicaciones se centran en cómo diseñar los sistemas para obtener una cierta funcionalidad y, al mismo tiempo, un diseño escalable.

4.4.1. Observer

El patrón *Observer* se utiliza para definir relaciones 1 a n de forma que un objeto pueda notificar y/o actualizar el estado de otros automáticamente.

Problema

Tal y como se describió en la sección 4.3.4, el dominio de la aplicación en el patrón MVC puede notificar de cambios a las diferentes vistas. Es importante que el dominio no conozca los tipos concretos de las vistas, de forma que haya que modificar el dominio en caso de añadir o quitar una vista.

Otros ejemplos donde se da este tipo de problemas ocurren cuando el estado un elemento tiene influencia directa sobre otros. Por ejemplo, si en el juego se lanza un misil seguramente haya que notificar que se ha producido el lanzamiento a diferentes sistemas de sonido, partículas, luz, y los objetos que puedan estar alrededor.

Solución

El patrón Observer proporciona un diseño con poco acoplamiento entre los **observadores** y el objeto **observado**. Siguiendo la filosofía de *publicación/suscripción*, los objetos observadores se deben registrar en el objeto observado, también conocido como *subject*. Así, cuando ocurra el evento oportuno, el subject recibirá una invocación a través de `notify()` y será el encargado de «notificar» a todos los elementos suscritos a él a través del método `update()`. Los observadores que reciben la invocación pueden realizar las acciones pertinentes como consultar el estado del dominio para obtener nuevos valores. En la figura 4.12 se muestra un esquema general del patrón Observer.

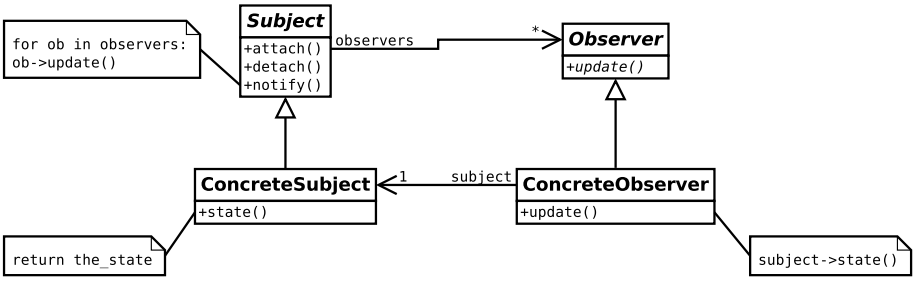


Figura 4.12: Diagrama de clases del patrón Observer

A modo de ejemplo, en la figura 4.13 se muestra un diagrama de secuencia en el que se describe el orden de las invocaciones en un sistema que utiliza el patrón Observer. Nótese que los observadores se suscriben al subject (`RocketLauncher`) y, a continuación, reciben las actualizaciones. También pueden dejar de recibirlas, utilizando la operación `detach()`.

Consideraciones

Al emplear el patrón Observer en nuestros diseños, se deben tener en cuenta las siguientes consideraciones:

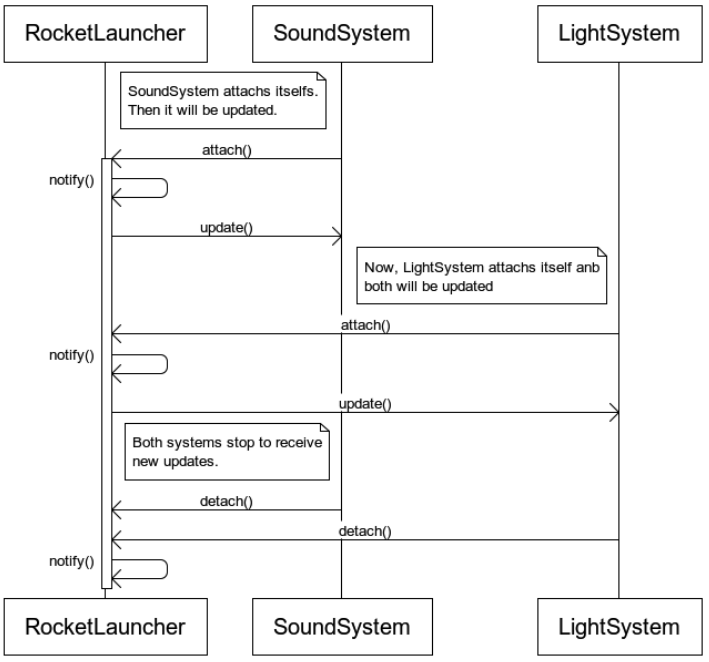


Figura 4.13: Diagrama de secuencia de ejemplo utilizando un Observer

- El objeto subject puede encapsular funcionalidad **compleja** semánticamente que será notificada asíncronamente a los observadores. El objeto observable, puede definir diferentes estrategias a la hora de notificar un cambio.
- `Subject` y sus observadores forman un modelo *push/pull*, por lo que evita la creación de protocolos de comunicación concretos entre ellos. Toda comunicación de este tipo pueden realizarse de la misma forma.
- No es necesario que el subject sea el que realiza la llamada a `notify()`. Un cliente externo puede ser el que fuerce dicha llamada.
- Los observadores pueden estar suscritos a más de un subject.
- Los observadores no tienen control sobre las actualizaciones no deseadas. Es posible que un observador no esté interesado en ciertas notificaciones y que sea necesario consultar al `Subject` por su estado en demasiadas ocasiones. Esto puede ser un problema en determinados escenarios.

Los *canales de eventos* es un patrón más genérico que el Observer pero que sigue respetando el modelo *push/pull*. Consiste en definir estructuras que permiten la comunicación n a n a través de un medio de comunicación (canal) que se puede multiplexar en diferentes temas (*topics*). Un objeto puede establecer un rol *suscriptor* de un tema dentro de un canal y sólo recibir las notificaciones del mismo. Además, también puede configurarse como *publicador*, por lo que podría enviar actualizaciones al mismo canal.

4.4.2. State

El patrón *State* es útil para realizar transiciones de estado e implementar autómatas respetando el principio de encapsulación.

Problema

Es muy común que en cualquier aplicación, incluído los videojuegos, existan estructuras que pueden ser modeladas directamente como un *autómata*, es decir, una colección de estados y unas transiciones dependientes de una entrada. En este caso, la entrada pueden ser invocaciones y/o eventos recibidos.

Por ejemplo, los estados de un personaje de un videojuego podrían ser: de pie, tumbado, andando y saltando. Dependiendo del estado en el que se encuentre y de la invocación recibida, el siguiente estado será uno u otro. Por ejemplo, si está de pie y recibe la orden de tumbarse, ésta se podrá realizar. Sin embargo, si ya está tumbado no tiene sentido volver a tumbarse, por lo que debe permanecer en ese estado.

Solución

El patrón State permite encapsular el mecanismo de las transiciones que sufre un objeto a partir de los estímulos externos. En la figura 4.14 se muestra un ejemplo de aplicación del mismo. La idea es crear una clase abstracta que representa al estado del personaje (`CharacterState`). En ella se definen las mismas operaciones que puede recibir el personaje con una implementación por defecto. En este caso, la implementación es vacía.

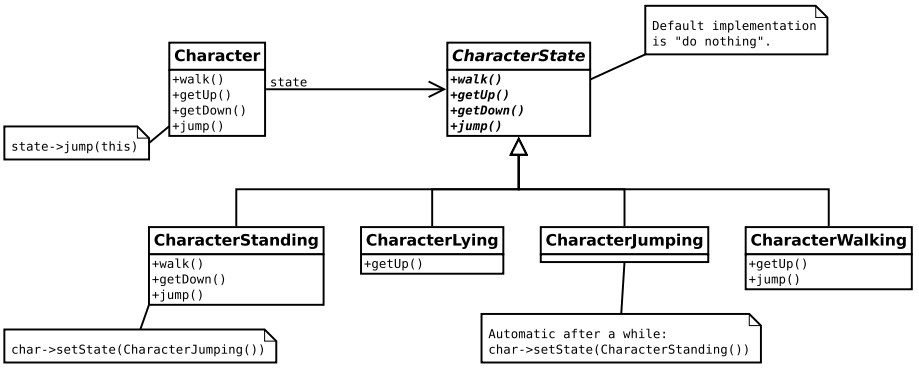


Figura 4.14: Ejemplo de aplicación del patrón State

Por cada estado en el que puede encontrarse el personaje, se crea una clase que hereda de la clase abstracta anterior, de forma que en cada una de ellas se implementen los métodos que producen cambio de estado.

Por ejemplo, según el diagrama, en el estado «de pie» se puede recibir la orden de caminar, tumbarse y saltar, pero no de levantarse. En caso de recibir esta última, se ejecutará la implementación por defecto, es decir, no hacer nada.

En definitiva, la idea es que las clases que representan a los estados sean las encargadas de cambiar el estado del personaje, de forma que los cambios de estados quedan encapsulados y delegados al estado correspondiente.

Consideraciones

- Los componentes del diseño que se comporten como autómatas son buenos candidatos a ser modelados con el patrón State. Una conexión TCP (Transport Control Protocol) o un carrito en una tienda web son ejemplos de este tipo de problemas.
- Es posible que una entrada provoque una situación de error estando en un determinado estado. Para ello, es posible utilizar las excepciones para notificar dicho error.
- Las clases que representan a los estados **no** deben mantener un estado intrínseco, es decir, no se debe hacer uso de variables que dependan de un contexto. De esta forma, el estado puede compartirse entre varias instancias. La idea de compartir un estado que no depende del contexto es la base fundamental del patrón *Flyweight*, que sirve para las situaciones en las que crear muchas instancias puede ser un problema de rendimiento.

4.4.3. Iterator

El patrón *Iterator* se utiliza para ofrecer una interfaz de acceso secuencial a una determinada estructura ocultando la representación interna y la forma en que realmente se accede.

Problema

Manejar colecciones de datos es algo muy habitual en el desarrollo de aplicaciones. Listas, pilas y, sobre todo, árboles son ejemplos de estructuras de datos muy presentes en los juegos y se utilizan de forma intensiva.

Una operación muy habitual es recorrer las estructuras para analizar y/o buscar los datos que contienen. Es posible que sea necesario recorrer la estructura de forma secuencial, de dos en dos o, simplemente, de forma aleatoria. Los clientes suelen implementar el método concreto con el que desean recorrer la estructura por lo que puede ser un problema si, por ejemplo, se desea recorrer una misma estructura de datos de varias formas distintas. Conforme aumenta las combinaciones entre los tipos de estructuras y métodos de acceso, el problema se agrava.

Solución

Con ayuda del patrón Iterator es posible obtener acceso secuencial, desde el punto de vista del usuario, a cualquier estructura de datos, independientemente de su implementación interna. En la figura 4.15 se muestra un diagrama de clases genérico del patrón. Como puede verse, la estructura de datos es la encargada de crear el iterador adecuado para ser accedida a través del método `iterator()`. Una vez que el cliente ha obtenido el iterador, puede utilizar los métodos de acceso que ofrecen tales como `next()` (para obtener el siguiente elemento) o `isDone()` para comprobar si no existen más elementos.

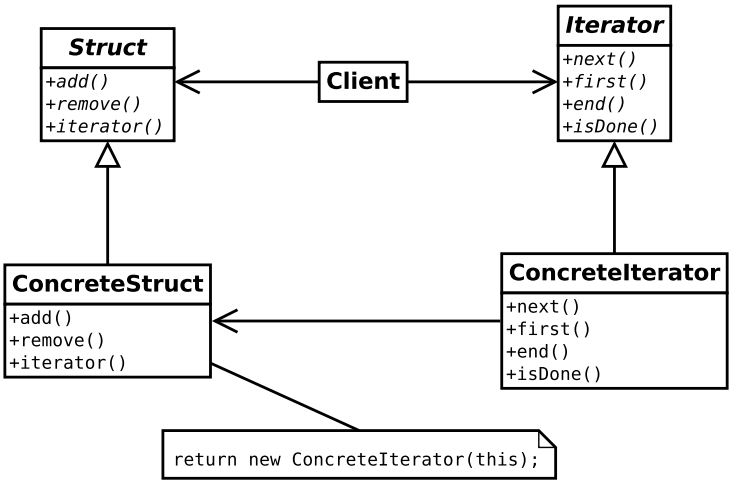


Figura 4.15: Diagrama de clases del patrón Iterator

Implementación

A continuación, se muestra una implementación simplificada y aplicada a una estructura de datos de tipo lista. Nótese cómo utilizando las primitivas que ofrece la estructura, el iterator proporciona una visión de acceso secuencial a través del método `next()`.

Listado 4.8: Iterator (ejemplo)

```
1  class List : public Struct {
2  public:
3    void add(const Object& ob) { /* add element in a list*/ };
4    void remove(const Object& ob) { /* remove element in a list*/ };
5    Object get_at(const int index) { /* get list[index] element*/ };
6
7    /* more access methods */
8
9    void iterator(const Object& ob) {
10     return new ListIterator(this);
11   };
12 };
13
14 class ListIterator : public Iterator {
15 private:
16   int _currentIndex;
17   List _list;
18
19 public:
20   ListIterator (List* list) : _currentIndex(0), _list(list) { };
21
22   Object next() {
23     if (isDone()) {
24       throw new IteratorOutOfBounds();
25     }
26     Object retval = _list->get_at(_currentIndex);
27     _currentIndex++;
28     return retval;
29   };
30
31   Object first() {
32     return _list->get_at(0);
33   };
34
```

```
35   bool isDone() {
36     return _currentIndex > _list->length();
37   };
38 };
39
40 /* client using iterator */
41
42 List list = new List();
43 ListIterator it = list.iterator();
44
45 for (Object ob = it.first(); not it.isDone(); it.next()) {
46   // do the loop using 'ob'
47 };
```

Consideraciones

La ventaja fundamental de utilizar el patrón Iterator es la simplificación de los clientes que acceden a las diferentes estructuras de datos. El control sobre el acceso lo realiza el propio iterador y, además, almacena todo el estado del acceso del cliente. De esta forma se crea un nivel de abstracción para los clientes que acceden siempre de la misma forma a cualquier estructura de datos con iteradores.

Obviamente, nuevos tipos de estructuras de datos requieren nuevos iteradores. Sin embargo, para añadir nuevos tipos de iteradores a estructuras ya existentes puede realizarse de dos formas:

- Añadir un método virtual en la clase padre de todas las estructuras de forma que los clientes puedan crear el nuevo tipo iterador. Esto puede tener un gran impacto porque puede haber estructuras que no se pueda o no se desee acceder con el nuevo iterador.
- Añadir un nuevo tipo de estructura que sea dependiente del nuevo tipo del iterador. Por ejemplo, `RandomizedList` que devolvería un iterador `RandomIterator` y que accede de forma aleatoria a todos los elementos.

La STL de C++ implementa el patrón Iterator en todos los contenedores que ofrece.

4.4.4. Template Method

El patrón *Template Method* se puede utilizar cuando es necesario redefinir algunos pasos de un determinado algoritmo utilizando herencia.

Problema

En un buen diseño los algoritmos complejos se dividen en funciones más pequeñas, de forma que si se llama a dichas funciones en un determinado orden se consigue implementar el algoritmo completo. Conforme se diseña cada paso concreto, se suele ir detectando funcionalidad común con otros algoritmos.

Por ejemplo, supongamos que tenemos dos tipos de jugadores de juegos de mesa: ajedrez y damas. En esencia, ambos juegan igual; lo que cambia son las reglas del juego que, obviamente, condiciona su estrategia y su forma de jugar concreta. Sin embargo, en ambos juegos, los jugadores mueven en su turno, esperan al rival y esto se repite hasta que acaba la partida.

El patrón Template Method consiste extraer este comportamiento común en una clase padre y definir en las clases hijas la funcionalidad concreta.

Solución

Siguiendo con el ejemplo de los jugadores de ajedrez y damas, la figura 4.16 muestra una posible aplicación del patrón Template Method a modo de ejemplo. Nótese que la clase `GamePlayer` es la que implementa el método `play()` que es el que invoca a los otros métodos que son implementados por las clases hijas. Este método es el *método plantilla*.

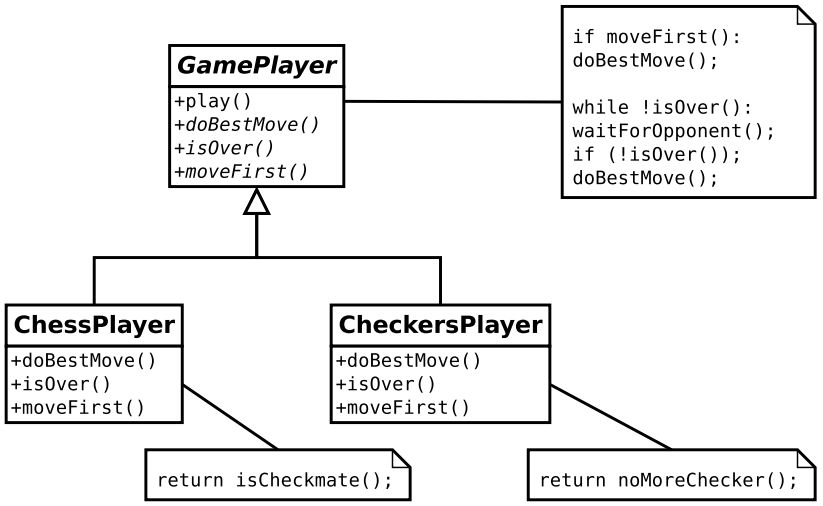


Figura 4.16: Aplicación de ejemplo del patrón Template Method

Cada tipo de jugador define los métodos en base a las reglas y heurísticas de su juego. Por ejemplo, el método `isOver()` indica si el jugador ya no puede seguir jugando porque se ha terminado el juego. En caso de las damas, el juego se acaba para el jugador si se ha quedado sin fichas; mientras que en el caso ajedrez puede ocurrir por jaque mate (además de otros motivos).

Consideraciones

Algunas consideraciones sobre el patrón Template Method:

- Utilizando este patrón se suele obtener estructuras altamente reutilizables.
- Introduce el concepto de operaciones *hook* que, en caso de no estar implementadas en las clases hijas, tienen una implementación por defecto. Las clases hijas pueden sobreescribirlas para añadir su propia funcionalidad.

4.4.5. Strategy

El patrón *Strategy* se utiliza para encapsular el funcionamiento de una familia de algoritmos, de forma que se pueda intercambiar su uso sin necesidad de modificar a los clientes.

Problema

En muchas ocasiones, se suele proporcionar diferentes algoritmos para realizar una misma tarea. Por ejemplo, el nivel de habilidad de un jugador viene determinado por diferentes algoritmos y heurísticas que determinan el grado de dificultad. Utilizando diferentes tipos algoritmos podemos obtener desde jugadores que realizan movimientos aleatorios hasta aquellos que pueden tener cierta inteligencia y que se basan en técnicas de IA.

Lo deseable sería poder tener jugadores de ambos tipos y que, desde el punto de vista del cliente, no fueran tipos distintos de jugadores. Simplemente se comportan diferente porque usan distintos algoritmos internamente, pero todos ellos son jugadores.

Solución

Mediante el uso de la herencia, el patrón Strategy permite encapsular diferentes algoritmos para que los clientes puedan utilizarlos de forma transparente. En la figura 4.17 puede verse la aplicación de este patrón al ejemplo anterior de los jugadores.

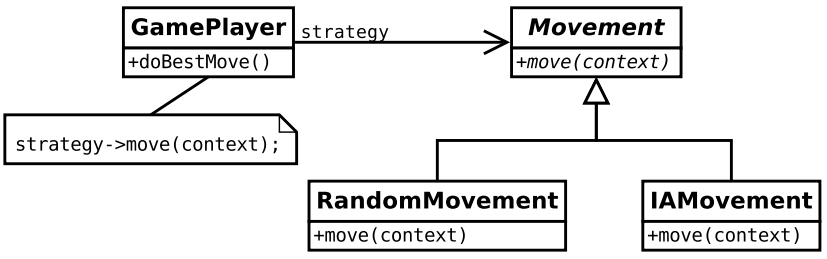


Figura 4.17: Aplicación de ejemplo del patrón Strategy

La idea es extraer los métodos que conforman el comportamiento que puede ser intercambiado y encapsularlo en una familia de algoritmos. En este caso, el movimiento del jugador se extrae para formar una jerarquía de diferentes movimientos (`Movement`). Todos ellos implementan el método `move()` que recibe un contexto que incluye toda la información necesaria para llevar a cabo el algoritmo.

El siguiente fragmento de código indica cómo se usa este esquema por parte de un cliente. Nótese que al configurarse cada jugador, ambos son del mismo tipo de cara al cliente aunque ambos se comportarán de forma diferente al invocar al método `doBestMove()`.

Listado 4.9: Uso de los jugadores (Strategy)

```
1 GamePlayer bad_player = new GamePlayer(new RandomMovement());
2 GamePlayer good_player = new GamePlayer(new IAMovement());
3
4 bad_player->doBestMove();
5 good_player->doBestMove();
```

Consideraciones

El patrón Strategy es una buena alternativa a realizar subclases en las entidades que deben comportarse de forma diferente en función del algoritmo utilizado. Al extraer la heurística a una familia de algoritmos externos, obtenemos los siguientes beneficios:

- Se aumenta la reutilización de dichos algoritmos.
- Se evitan sentencias condicionales para elegir el comportamiento deseado.
- Los clientes pueden elegir diferentes implementaciones para un mismo comportamiento deseado, por lo que es útil para depuración y pruebas donde se pueden escoger implementaciones más simples y rápidas.

4.4.6. Reactor

El patrón *Reactor* es un patrón arquitectural para resolver el problema de cómo atender peticiones concurrentes a través de señales y manejadores de señales.

Problema

Existen aplicaciones, como los servidores web, cuyo comportamiento es *reactivo*, es decir, a partir de la ocurrencia de un evento externo se realizan todas las operaciones necesarias para atender a ese evento externo. En el caso del servidor web, una conexión entrante (evento) dispararía la ejecución del código pertinente que crearía un hilo de ejecución para atender a dicha conexión. Pero también pueden tener comportamiento *proactivo*. Por ejemplo, una señal interna puede indicar cuándo destruir una conexión con un cliente que lleva demasiado tiempo sin estar accesible.

En los videojuegos ocurre algo muy similar: diferentes entidades pueden lanzar eventos que deben ser tratados en el momento en el que se producen. Por ejemplo, la pulsación de un botón en el joystick de un jugador es un evento que debe ejecutar el código pertinente para que la acción tenga efecto en el juego.

Solución

En el patrón Reactor se definen una serie de actores con las siguientes responsabilidades (véase figura 4.18):

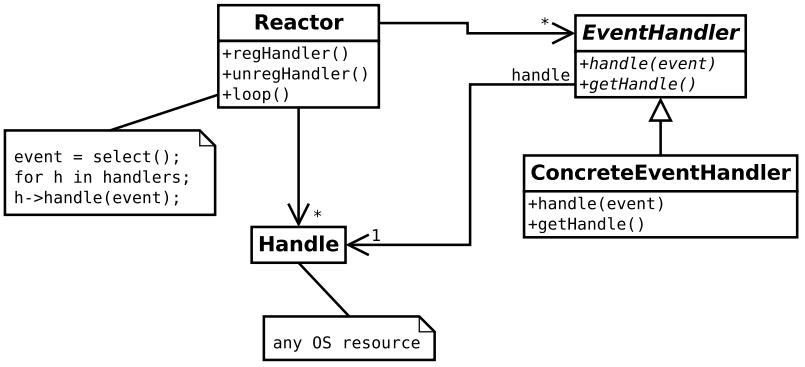


Figura 4.18: Diagrama de clases del patrón Reactor

- **Eventos**: los eventos externos que puedan ocurrir sobre los recursos (`Handles`). Normalmente su ocurrencia es asíncrona y siempre está relaciona a un recurso determinado.
- **Recursos** (`Handles`): se refiere a los objetos sobre los que ocurren los eventos.

  La pulsación de una tecla, la expiración de un temporizador o una conexión entrante en un socket son ejemplos de eventos que ocurren sobre ciertos recursos. La representación de los recursos en sistemas tipo GNU/Linux es el descriptor de fichero.
- **Manejadores de Eventos**: Asociados a los recursos y a los eventos que se producen en ellos, se encuentran los manejadores de eventos (`EventHandler`) que reciben una invocación a través del método `handle()` con la información del evento que se ha producido.
- **Reactor**: se trata de la clase que encapsula todo el comportamiento relativo a la desmultiplexación de los eventos en manejadores de eventos (*dispatching*). Cuando ocurre un cierto evento, se busca los manejadores asociados y se les invoca el método `handle()`.

En general, el comportamiento sería el siguiente:

1. Los manejadores se registran utilizando el método `regHandler()` del Reactor. De esta forma, el Reactor puede configurarse para esperar los eventos del recurso que el manejador espera. El manejador puede dejar de recibir notificaciones con `unregHandler()`.
2. A continuación, el Reactor entra en el bucle infinito (`loop()`), en el que se espera la ocurrencia de eventos.
3. Utilizando alguna llamada al sistema, como puede ser `select()`, el Reactor espera a que se produzca algún evento sobre los recursos monitorizados.
4. Cuando ocurre, busca los manejadores asociados a ese recurso y les invoca el método `handle()` con el evento que ha ocurrido como parámetro.
5. El manejador recibe la invocación y ejecuta todo el código asociado al evento.

Nótese que aunque los eventos ocurran concurrentemente el Reactor *serializa* las llamadas a los manejadores. Por lo tanto, la ejecución de los manejadores de eventos ocurre de forma secuencial.

Consideraciones

Al utilizar un Reactor, se deben tener las siguientes consideraciones:

1. Los manejadores de eventos no pueden consumir mucho tiempo. Si lo hacen, pueden provocar un efecto *convoy* y, dependiendo de la frecuencia de los eventos, pueden hacer que el sistema sea inoperable. En general, cuanto mayor sea la frecuencia en que ocurren los eventos, menos tiempo deben consumir los manejadores.
2. Existen implementaciones de Reactors que permiten una desmultiplexación concurrente.
3. Desde un punto de vista general, el patrón Observer tiene un comportamiento muy parecido. Sin embargo, el Reactor está pensado para las relaciones 1 a 1 y no 1 a n como en el caso del Observer.

4.4.7. Visitor

El patrón *Visitor* proporciona un mecanismo para realizar diferentes operaciones sobre una jerarquía de objetos de forma que añadir nuevas operaciones no haga necesario cambiar las clases de los objetos sobre los que se realizan las operaciones.

Problema

En el diseño de un programa, normalmente se obtienen jerarquías de objetos a través de herencia o utilizando el patrón Composite (véase sección 4.3.1). Considerando una jerarquía de objetos que sea más o menos estable, es muy probable que necesitemos realizar operaciones sobre dicha jerarquía. Sin embargo, puede ser que cada objeto deba ser tratado de una forma diferente en función de su tipo. La complejidad de estas operaciones aumenta muchísimo.

Supongamos el problema de detectar las colisiones entre los objetos de un juego. Dada una estructura de objetos (con un estado determinado), una primera aproximación sería recorrer toda la estructura en busca de dichas colisiones y, en cada caso particular, realizar las operaciones específicas que el objeto concreto necesita. Por ejemplo, en caso de detectarse una colisión de un misil con un edificio se producirán una serie de acciones diferentes a si el misil impacta contra un vehículo.

En definitiva, realizar operaciones sobre una estructura de objetos que mantiene un cierto estado puede complicar la implementación de las mismas debido a que se deben de tener en cuenta las particularidades de cada tipo de objeto y operación realizada.

Solución

El patrón Visitor se basa en la creación de dos jerarquías independientes:

- **Visitables**: son los elementos de la estructura de objetos que *aceptan* a un determinado visitante y que le proporcionan toda la información a éste para realizar una determinada operación.
- **Visitantes**: jerarquía de objetos que realizan una operación determinada sobre dichos elementos.

En la figura 4.19 se puede muestra un diagrama de clases genérico del patrón Visitor donde se muestran estas dos jerarquías. Cada visitante concreto realiza una operación sobre la estructura de objetos. Es posible que al visitante no le interesen todos los objetos y, por lo tanto, la implementación de alguno de sus métodos sea vacía. Sin embargo, lo importante del patrón Visitor es que se pueden añadir nuevos tipos de visitantes concretos y, por lo tanto, realizar nuevas operaciones sobre la estructura sin la necesidad de modificar nada en la propia estructura.

Implementación

Como ejemplo de implementación supongamos que tenemos una escena (`Scene`) en la que existe una colección de elementos de tipo `ObjectScene`. Cada elemento tiene atributos como su nombre, peso y posición en la escena, es decir, `name`, `weight` y `position`, respectivamente. Se definen dos tipos visitantes:

- `NameVisitor`: mostrará los nombres de los elementos de una escena.

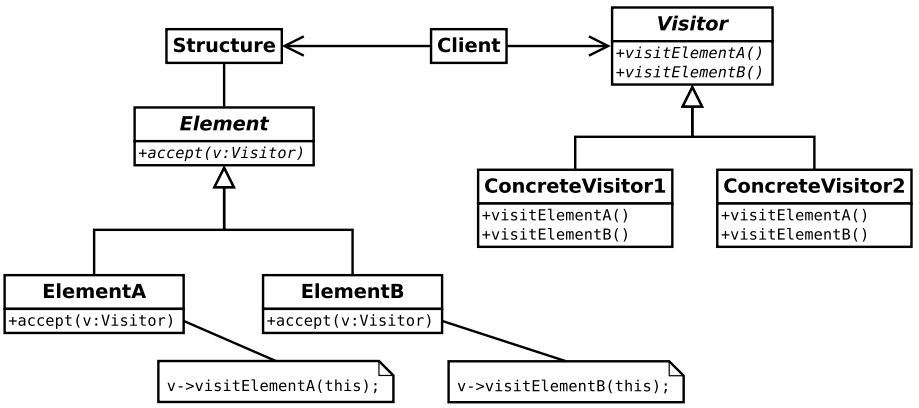


Figura 4.19: Diagrama de clases del patrón Visitor

- `BombVisitor`: modificará la posición final de todos los elementos de una escena al estallar una bomba. Para calcularla tendrá que tener en cuenta los valores de los atributos de cada objeto de la escena.

Se ha simplificado la implementación de `Scene` y `ObjectScene`. Únicamente se ha incluido la parte relativa al patrón visitor, es decir, la implementación de los métodos `accept()`. Nótese que es la escena la que ejecuta `accept()` sobre todos sus elementos y cada uno de ellos invoca a `visitObject()`, con una referencia a sí mismos para que el visitante pueda extraer información. Dependiendo del tipo de visitor instanciado, uno simplemente almacenará el nombre del objeto y el otro calculará si el objeto debe moverse a causa de una determinada explosión. Este mecanismo se conoce como despachado doble o *double dispatching*. El objeto que recibe la invocación del `accept()` delega la implementación de lo que se debe realizar a un tercero, en este caso, al visitante.

Finalmente, la escena también invoca al visitante para que realice las operaciones oportunas una vez finalizado el análisis de cada objeto. Nótese que, en el ejemplo, en el caso de `BombVisitor` no se realiza ninguna acción en este caso.

Listado 4.10: Visitor (ejemplo)

```
1  class ObjectScene {
2  public:
3    void accept(SceneVisitor* v) { v->visitObject(this); };
4  };
5
6  class Scene {
7  private:
8    vector<ObjectScene> _objects;
9  public:
10   void accept(SceneVisitor* v) {
11     for (vector<ObjectScene>::iterator ob = _objects.begin();
12      ob != _objects.end(); ob++)
13       v->accept(v);
14     v->visitScene(this);
15   };
16 };
17
18 class SceneVisitor {
19   virtual void visitObject(ObjectScene* ob) = 0;
20   virtual void visitScene(Scene* scene) = 0;
21 };
22
```

```
23 class NameVisitor : public SceneVisitor {
24 private:
25   vector<string> _names;
26 public:
27   void visitObject(ObjectScene* ob) {
28     _names.push_back(ob->name);
29   };
30   void visitScene(Scene* scene) {
31     cout << "The scene '" << scene->name << "' has following
           objects:"
32      << endl;
33     for (vector<string>::iterator it = _names.begin();
34      it != _names.end(); it++)
35       cout << *it << endl;
36   };
37 };
38
39 class BombVisitor : public SceneVisitor {
40 private:
41   Bomb _bomb;
42 public:
43   BombVisitor(Bomb bomb) : _bomb(bomb);
44   void visitObject(ObjectScene* ob) {
45     Point new_pos = calculateNewPosition(ob->position,
46                       ob->weight,
47                       _bomb->intensity);
48     ob->position = new_pos;
49   };
50   void visitScene(ObjectScene* scene) {};
51 };
52
53 /* client usage */
54 Scene* scene = createScene();
55 SceneVisitor* name_visitor = new NameVisitor();
56 scene->accept(name_visitor);
57 ...
58 /* bomb explosion occurs */
59 SceneVisitor* bomb_visitor = new BombVisitor(bomb);
60 scene->accept(bomb_visitor);
```

Consideraciones

Algunas consideraciones sobre el patrón Visitor:

- El patrón Visitor es muy conveniente para recorrer estructuras arbóreas y realizar operaciones en base a los datos almacenados.
- En el ejemplo, la ejecución se realiza de forma secuencial ya que se utiliza un iterador de la clase `vector`. La forma en que se recorra la estructura influirá notablemente en el rendimiento del análisis de la estructura. Se puede hacer uso del patrón Iterator para decidir cómo escoger el siguiente elemento.
- Uno de los problemas de este patrón es que no es recomendable si la estructura de objetos cambia frecuentemente o es necesario añadir nuevos tipos de objetos de forma habitual. Cada nuevo objeto que sea susceptible de ser visitado puede provocar grandes cambios en la jerarquía de los visitantes.

4.5. *Programming Idioms*

Hasta el momento, se han descrito algunos de los patrones de diseño más importantes que proporcionan una buena solución a determinados problemas a nivel de diseño. Todos ellos son aplicables a cualquier lenguaje de programación en mayo o menor medida: algunos patrones de diseño son más o menos difíciles de implementar en un lenguaje de programación determinado. Dependerá de las estructuras de abstracción que éste proporcione.

Sin embargo, a nivel de lenguaje de programación, existen «patrones» que hacen que un programador comprenda mejor dicho lenguaje y aplique soluciones mejores, e incluso óptimas, a la hora de resolver un problema de codificación. Los expresiones idiomáticas (o simplemente, *idioms*) son un conjunto de buenas soluciones de programación que permiten:

- Resolver ciertos problemas de codificación, normalmente asociados a un lenguaje específico. Por ejemplo, cómo obtener la dirección de memoria real de un objeto en C++ aunque esté sobreescrito el operador `&`. Este *idiom* se llama *AddressOf*.
- Entender y explotar las interioridades del lenguaje y, por tanto, programar mejor.
- Establecer un repertorio de buenas prácticas de programación para el lenguaje.

Al igual que los patrones, los *idioms* tienen un nombre asociado (además de sus alias), la definición del problema que resuelven y bajo qué contexto, así como algunas consideraciones (eficiencia, etc.). A continuación, se muestran algunos de los más relevantes. En la sección de «Patrones de Diseño Avanzados» se exploran más de ellos.

4.5.1. Orthodox Canonical Form

Veamos un ejemplo de mal uso de C++:

Listado 4.11: Ejemplo de uso incorrecto de C++

```
1  #include <vector>
2
3  struct A {
4    A() : a(new char[3000]) {}
5    ~A() { delete [] a; }
6    char* a;
7  };
8
9  int main() {
10   A var;
11   std::vector<A> v;
12   v.push_back(var);
13   return 0;
14 }
```

Si compilamos y ejecutamos este ejemplo nos llevaremos una desagradable sorpresa.

```
$ g++ bad.cc -o bad
$ ./bad
*** glibc detected *** ./bad: double free or corruption (!prev): 0
    x00000000025de010 ***
```

```
======= Backtrace: =========
...
```

¿Qué es lo que ha pasado? ¿No estamos reservando memoria en el constructor y liberando en el destructor? ¿Cómo es posible que haya corrupción de memoria? La solución al enigma es lo que no se ve en el código. Si no lo define el usuario el compilador de C++ añade automáticamente un constructor de copia que implementa la estrategia más simple, copia de todos los miembros. En particular cuando llamamos a `push_back()` creamos una copia de `var`. Esa copia recibe a su vez una copia del miembro `var.a` que es un puntero a memoria ya reservada. Cuando se termina `main()` se llama al destructor de `var` y del `vector`. Al destruir el vector se destruyen todos los elementos. En particular se destruye la copia de `var`, que a su vez libera la memoria apuntada por su miembro `a`, que apunta a la misma memoria que ya había liberado el destructor de `var`.

Antes de avanzar más en esta sección conviene formalizar un poco la estructura que debe tener una clase en C++ para no tener sorpresas. Básicamente se trata de especificar todo lo que debe implementar una clase para poder ser usada como un tipo cualquiera:

- Pasarlo como parámetro por valor o como resultado de una función.
- Crear *arrays* y contenedores de la STL.
- Usar algoritmos de la STL sobre estos contenedores.

Para que no aparezcan sorpresas una clase no trivial debe tener como mínimo:

1. *Constructor por defecto*. Sin él sería imposible instanciar *arrays* y no funcionarían los contenedores de la STL.
2. *Constructor de copia*. Sin él no podríamos pasar argumentos por valor, ni devolverlo como resultado de una función.
3. *Operador de asignación*. Sin él no funcionaría la mayoría de los algoritmos sobre contenedores.
4. *Destructor*. Es necesario para liberar la memoria dinámica reservada. El destructor por defecto puede valer si no hay reserva explícita.

A este conjunto de reglas se le llama normalmente forma canónica ortodoxa (*orthodox canonical form*).

Además, si la clase tiene alguna función virtual, el destructor debe ser virtual. Esto es así porque si alguna función virtual es sobrecargada en clases derivadas podrían reservar memoria dinámica que habría que liberar en el destructor. Si el destructor no es virtual no se podría garantizar que se llama. Por ejemplo, porque la instancia está siendo usada a través de un puntero a la clase base.

4.5.2. Interface Class

En C++, a diferencia de lenguajes como Java, no existen clases interfaces, es decir, tipos abstractos que no proporcionan implementación y que son muy útiles para definir el contrato entre un objeto cliente y uno servidor. Por ello, el concepto de interfaz es muy importante en POO. Permiten obtener un bajo grado de acoplamiento entre las entidades y facilitan en gran medida las pruebas unitarias, ya que permite utilizar objetos dobles (mocks, stubs, etc.) en lugar de las implementaciones reales.

Listado 4.12: Ejemplo de Interface Class

```
1  class Vehicle
2  {
3  public:
4    virtual ~Vehicle();
5    virtual std::string name() = 0;
6    virtual void run() = 0;
7  };
8
9  class Car: public Vehicle
10 {
11 public:
12   virtual ~Car();
13   std::string name() { return "Car"; }
14   void run() { /* ... */ }
15 };
16
17 class Motorbike: public Vehicle
18 {
19 public:
20   virtual ~Motorbike();
21   std::string name() { return "Motorbike"; }
22   void run() { /* ... */ }
23 };
```

Este *idiom* muestra cómo crear una clase que se comporta como una interfaz, utilizando métodos*virtuales puros*.

El compilador fuerza que los métodos virtuales puros sean implementados por las clases derivadas, por lo que fallará en tiempo de compilación si hay alguna que no lo hace. Como resultado, tenemos que `Vehicle` actúa como una clase interfaz. Nótese que el destructor se ha declarado como virtual. Como ya se citó en la forma canónica ortodoxa (sección 4.5.1), esto es una buena práctica para evitar posibles *leaks* de memoria en tiempo de destrucción de un objeto `Vehicle` usado polimórficamente (por ejemplo, en un contenedor). Con esto, se consigue que se llame al destructor de la clase más derivada.

4.5.3. Final Class

En Java o C# es posible definir una clase como `final`, es decir, no es posible heredar una clase hija de ella. Este mecanismo de protección puede tener mucho sentido en diferentes ocasiones:

- Desarrollo de una librería o una API que va ser utilizada por terceros.
- Clases externas de módulos internos de un programa de tamaño medio o grande.

En C++, no existe este mecanismo. Sin embargo, es posible simularlo utilizando el *idiom* Final Class.

Este *idiom* se basa en una regla de la herencia virtual: el constructor y destructor de una clase heredada virtualmente será invocado por la clase más derivada de la jerarquía. Como, en este caso, el destructor es privado, el compilador prohíbe la instanciación de `B` y el efecto es que no se puede heredar más allá de `A`.

Listado 4.13: Ejemplo de Final Class

```
1  class Final
2  {
3    ~Final() {} // privado
4    friend class A;
5  };
6
7  class A : virtual Final
8  { };
9
10 class B : public B
11 { };
12
13 int main (void)
14 {
15   B b; // fallo de compilación
16 }
```

4.5.4. pImpl

Pointer To Implementation (pImpl), también conocido como *Handle Body* u *Opaque Pointer*, es un famoso *idiom* (utilizado en otros muchos) para ocultar la implementación de una clase en C++. Este mecanismo puede ser muy útil sobre todo cuando se tienen componentes reutilizables cuya declaración o interfaz puede cambiar, lo cual implicaría recompilar a todos sus usuarios. El objetivo es minimizar el impacto de un cambio en la declaración de la clase a sus usuarios.

En C++, un cambio en las variables miembro de una clase o en los métodos `inline` puede suponer que los usuarios de dicha clase tengan que recompilar. Para resolverlo, la idea de pImpl es que la clase ofrezca una interfaz pública *bien definida* y que ésta contenga un puntero a su implementación, descrita de forma privada.

Por ejemplo, la clase `Vehicle` podría ser de la siguiente forma:

Listado 4.14: Ejemplo básico de clase Vehicle

```
1 class Vehicle
2 {
3 public:
4   void run(int distance);
5
6 private:
7   int _wheels;
8 };
```

Como se puede ver, es una clase muy sencilla ya que ofrece sólo un método público. Sin embargo, si queremos modificarla añadiendo más atributos o nuevos métodos privados, se obligará a los usuarios de la clase a recompilar por algo que realmente no utilizan directamente. Usando pImpl, quedaría el siguiente esquema:

Listado 4.15: Clase Vehicle usando pImpl (Vehicle.h)

```
1 /* Interfaz publica Vehicle.h */
2
3 class Vehicle
4 {
5 public:
```

```
6   void run(int distance);
7
8  private:
9   class VehicleImpl;
10  VehicleImpl* _pimpl;
11 };
```

Listado 4.16: Implementación de Vehicle usando pImpl (Vehicle.cpp

```
1  /* Vehicle.cpp */
2
3  #include <Vehicle.h>
4
5  Vehicle::Vehicle()
6  {
7    _pimpl = new VehicleImpl();
8  }
9
10 void Vehicle::run()
11 {
12   _pimpl->run();
13 }
```

Listado 4.17: Declaración de VehicleImpl (VehicleImpl.h

```
1  /* VehicleImpl.h */
2
3  class VehicleImpl
4  {
5  public:
6    void run();
7
8  private:
9    int _wheels;
10   std::string name;
11 };
```

Si la clase Vehicle está bien definida, se puede cambiar su implementación sin obligar a sus usuarios a recompilar. Esto proporciona una mayor flexibilidad a la hora de definir, implementar y realizar cambios sobre la clase `VehicleImpl`.

Capítulo 5

La Biblioteca STL

David Vallejo Fernández

capitalEn este capítulo se proporciona una visión general de STL (*Standard Template Library*), la biblioteca estándar proporcionada por C++, en el contexto del desarrollo de videojuegos, discutiendo su utilización en dicho ámbito de programación. Asimismo, también se realiza un recorrido exhaustivo por los principales tipos de contenedores, estudiando aspectos relevantes de su implementación, rendimiento y uso en memoria. El objetivo principal que se pretende alcanzar es que el lector sea capaz de **utilizar la estructura de datos más adecuada** para solucionar un problema, justificando el porqué y el impacto que tiene dicha decisión sobre el proyecto en su conjunto.

5.1. Visión general de STL

Usando STL

STL es sin duda una de las bibliotecas más utilizadas en el desarrollo de aplicaciones de C++. Además, está muy optimizada para el manejo de estructuras de datos y de algoritmos básicos, aunque su complejidad es elevada y el código fuente es poco legible para desarrolladores poco experimentados.

Desde un punto de vista abstracto, la **biblioteca estándar de C++**, STL, es un conjunto de clases que proporciona la siguiente funcionalidad [94]:

- Soporte a características del lenguaje, como por ejemplo la gestión de memoria e información relativa a los tipos de datos manejados en tiempo de ejecución.
- Soporte relativo a aspectos del lenguaje definidos por la implementación, como por ejemplo el valor en punto flotante con mayor precisión.
- Soporte para funciones que no se pueden implementar de manera óptima con las herramientas del propio lenguaje, como por ejemplo ciertas funciones matemáticas o asignación de memoria dinámica.

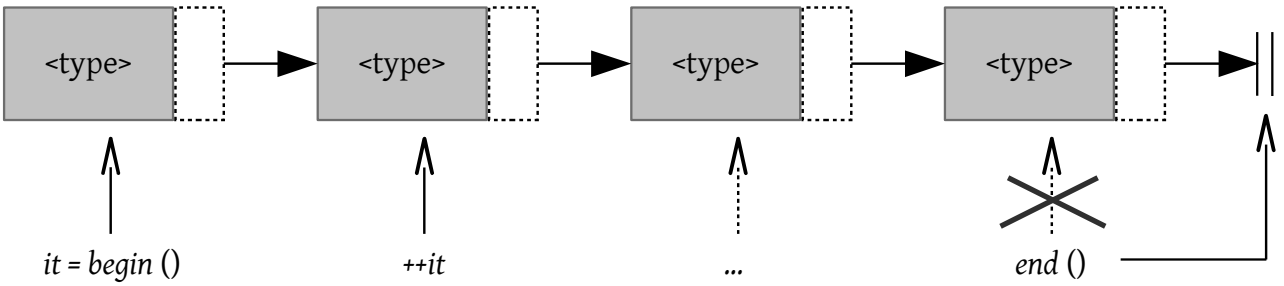


Figura 5.1: Representación gráfica del recorrido de un contenedor mediante iteradores.

- Inclusión de contenedores para almacenar datos en distintas estructuras, iteradores para acceder a los elementos de los contenedores y algoritmos para llevar a cabo operaciones sobre los mismos.
- Soporte como base común para otras bibliotecas.

La figura 5.2 muestra una perspectiva global de la **organización de STL** y los diferentes elementos que la definen. Como se puede apreciar, STL proporciona una gran variedad de elementos que se pueden utilizar como herramientas para la resolución de problemas dependientes de un dominio en particular.

Las utilidades de la biblioteca estándar se definen en el espacio de nombres *std* y se encuentran a su vez en una serie de bibliotecas, que identifican las partes fundamentales de STL. Note que no se permite la modificación de la biblioteca estándar y no es aceptable modificar su contenido mediante macros, ya que afectaría a la portabilidad del código desarrollado.

En el ámbito del desarrollo de videojuegos, los **contenedores** juegan un papel fundamental como herramienta para el almacenamiento de información en memoria. En este contexto, la realización un estudio de los mismos, en términos de operaciones, gestión de memoria y rendimiento, es especialmente importante para utilizarlos adecuadamente. Los contenedores están representados por dos tipos principales. Por una parte, las **secuencias** permiten almacenar elementos en un determinado orden. Por otra parte, los **contenedores asociativos** no tienen vinculado ningún tipo de restricción de orden.

Abstracción STL

El uso de los iteradores en STL representa un mecanismo de abstracción fundamental para realizar el recorrido, el acceso y la modificación sobre los distintos elementos almacenados en un contenedor.

La herramienta para recorrer los contenedores está representada por el **iterador**. Todos los contenedores mantienen dos funciones relevantes que permiten obtener dos iteradores:

1. *begin()*, que devuelve un iterador al primer elemento del contenedor,
2. *end()*, que devuelve un iterador al elemento siguiente al último.

El hecho de que *end()* devuelva un iterador al siguiente elemento al último albergado en el contenedor es una convención en STL que simplifica la iteración sobre los elementos del mismo o la implementación de algoritmos.

Una vez que se obtiene un iterador que apunta a una parte del contenedor, es posible utilizarlo como referencia para acceder a los elementos contiguos. En función del tipo de iterador y de la funcionalidad que implemente, será posible acceder solamente al elemento contiguo, al contiguo y al anterior o a uno aleatorio.

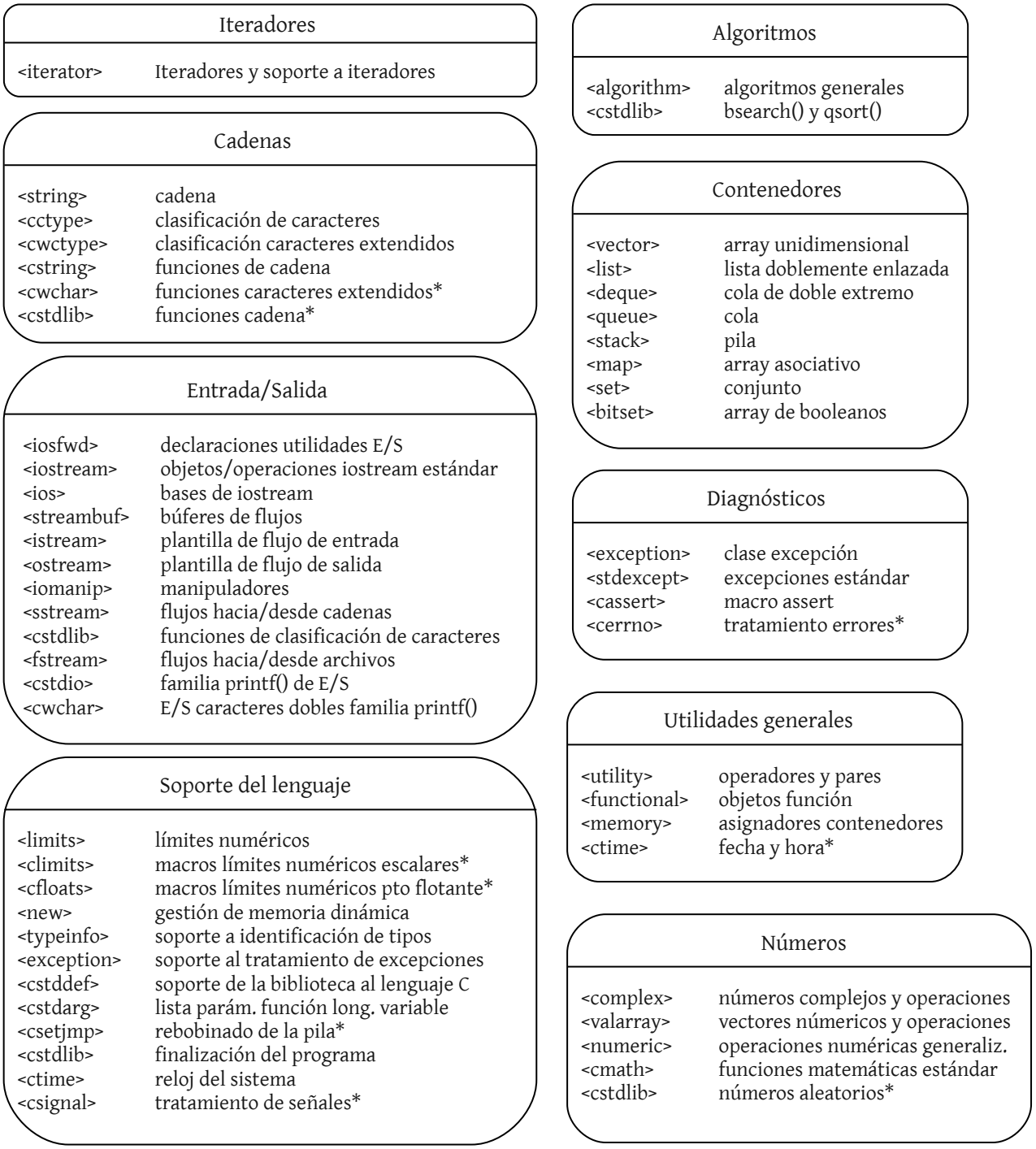


Figura 5.2: Visión general de la organización de STL [94]. El asterisco referencia a elementos con el estilo del lenguaje C.

El siguiente fragmento de código muestra un **ejemplo sencillo** en el que se utiliza STL para instanciar un vector de enteros, añadir elementos y, finalmente, recorrerlo haciendo uso de un iterador. Como se puede apreciar en la línea (7), STL hace uso de plantillas para manejar los distintos contenedores, permitiendo separar la funcionalidad de los mismos respecto a los datos que contienen. En el ejemplo, se instancia un vector de tipos enteros.

Más adelante se estudirán los distintos tipos de contenedores y la funcionalidad que ofrecen. Sin embargo, el lector puede suponer que cada contenedor proporciona una funcionalidad distinta en función de su naturaleza, de manera que cada uno de ellos ofrece una API que se utilizará para acceder a la misma. En el ejemplo, se utiliza en concreto la función *push_back* para añadir un elemento al final del vector.

Finalmente, en la línea (13) se declara un iterador de tipo *vector<int>* que se utilizará como base para el recorrido del vector. La inicialización del mismo se encuentra en la línea (15), es decir, en la parte de inicialización del bucle *for*, de manera que originalmente el iterador apunta al primer elemento del vector. La iteración sobre el vector se estará realizando hasta que no se llegue al iterador devuelto por *end()* (línea (16)), es decir, al elemento posterior al último.

Manejo de iteradores

El concepto de iterador es fundamental para recorrer e interactuar de manera eficiente con las distintas estructuras de datos incluidas en la biblioteca estándar.

Listado 5.1: Ejemplo sencillo de uso de STL

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  int main () {
7    vector<int> v;   // Instanciar el vector de int.
8
9    v.push_back(7);  // Añadir información.
10   v.push_back(4);
11   v.push_back(6);
12
13   vector<int>::iterator it;  // Declarar el iterador.
14
15   for (it = v.begin(); // it apunta al principio del vector,
16        it != v.end();  // mientras it no llegue a end(),
17        ++it)           // incrementar el iterador.
18     cout << *it << endl;  // Acceso al contenido de it.
19
20   return 0;
21 }
```

Note cómo el incremento del iterador es trivial (línea (17)), al igual que el acceso del contenido al que apunta el iterador (línea (18)), mediante una nomenclatura similar a la utilizada para manejar punteros.

Por otra parte, STL proporciona un conjunto estándar de **algoritmos** que se pueden aplicar a los contenedores e iteradores. Aunque dichos algoritmos son básicos, éstos se pueden combinar para obtener el resultado deseado por el propio programador. Algunos ejemplos de algoritmos son la búsqueda de elementos, la copia, la ordenación, etc. Al igual que ocurre con todo el código de STL, los algoritmos están muy optimizados y suelen sacrificar la simplicidad para obtener mejores resultados que proporcionen una mayor eficiencia.

5.2. STL y el desarrollo de videojuegos

En la sección 1.2 se discutió una visión general de la arquitectura estructurada en capas de un motor de juegos. Una de esas capas es la que proporciona bibliotecas de desarrollo, herramientas transversales y *middlewares* con el objetivo de dar soporte al proceso de desarrollo de un videojuego (ver sección 1.2.2). STL estaría incluido en dicha capa como biblioteca estándar de C++, posibilitando el uso de diversos contenedores como los mencionados anteriormente.

Desde un punto de vista general, a la hora de abordar el desarrollo de un videojuego, el programador o ingeniero tendría que decidir si utilizar STL o, por el contrario, utilizar alguna otra biblioteca que se adapte mejor a los requisitos impuestos por el juego a implementar.

5.2.1. Reutilización de código

Uno de los principales argumentos para utilizar STL en el desarrollo de videojuegos es la reutilización de código que ya ha sido implementado, depurado y portado a distintas plataformas. En este contexto, STL proporciona directamente estructuras de datos y algoritmos que se pueden utilizar y aplicar, respectivamente, para implementar soluciones dependientes de un dominio, como por ejemplo los videojuegos.

Gran parte del código de STL está vinculado a la construcción de bloques básicos en el desarrollo de videojuegos, como las listas, las tablas *hash*, y a algoritmos fundamentales como la ordenación o la búsqueda de elementos. Además, el diseño de STL está basado en el **uso extensivo de plantillas**. Por este motivo, es posible utilizarlo para manejar cualquier tipo de estructura de datos sin tener que modificar el diseño interno de la propia aplicación.

Figura 5.3: La reutilización de código es esencial para agilizar el desarrollo y mantenimiento de programas.

Recuerde que la reutilización de código es uno de los pilares fundamentales para afrontar el desarrollo de proyectos complejos y de gran envergadura, como por ejemplo los videojuegos.

Otra ventaja importante de STL es que muchos desarrolladores y programadores de todo el mundo lo utilizan. Este hecho tiene un impacto enorme, ya que ha posibilitado la creación de una **comunidad** a nivel mundial y ha afectado al diseño y desarrollo de bibliotecas utilizadas para programar en C++. Por una parte, si alguien tiene un problema utilizando STL, es relativamente fácil buscar ayuda y encontrar la solución al problema, debido a que es muy probable que alguien haya tenido ese mismo problema con anterioridad. Por otra parte, los desarrolladores de bibliotecas implementadas en C++ suelen tener en cuenta el planteamiento de STL en sus propios proyectos para facilitar la interoperabilidad, facilitando así el uso de dichas bibliotecas y su integración con STL.

5.2.2. Rendimiento

Uno de los aspectos críticos en el ámbito del desarrollo de videojuegos es el rendimiento, ya que es fundamental para lograr un sensación de interactividad y para dotar de sensación de realismo el usuario de videojuegos. Recuerde que un videojuego es una aplicación gráfica de renderizado en tiempo real y, por lo tanto, es necesario asegurar una tasa de *frames* por segundo adecuada y en todo momento. Para ello, el rendimiento de la aplicación es crítico y éste viene determinado en gran medida por las herramientas utilizadas.

En general, STL proporciona un muy buen rendimiento debido principalmente a que ha sido mejorado y optimizado por cientos de desarrolladores en estos últimos años, considerando las propiedades intrínsecas de cada uno de sus elementos y de las plataformas sobre las que se ejecutará.

La herramienta ideal

Benjamin Franklin afirmó que si dispusiera de 8 horas para derribar un árbol, emplearía 6 horas para afilar su hacha. Esta reflexión se puede aplicar perfectamente a la hora de decidir qué estructura de datos utilizar para solventar un problema.

STL será normalmente más eficiente que otra implementación y, por lo tanto, es el **candidato ideal** para manejar las estructuras de datos de un videojuego. Mejorar el rendimiento de STL implica tener un conocimiento muy profundo de las estructuras de datos a manejar y puede ser una alternativa en casos extremos y con plataformas hardware específicas. Si éste no es el caso, entonces STL es la alternativa directa.

No obstante, algunas compañías tan relevantes en el ámbito del desarrollo de videojuegos, como EA (Electronic Arts), han liberado su propia adaptación de STL denominada EASTL (Electronic Arts Standard Template Library) [1], justificando esta decisión en base a la detección de ciertas debilidades, como el modelo de asignación de memoria, o el hecho de garantizar la consistencia desde el punto de vista de la portabilidad.

Además del rendimiento de STL, es importante considerar que su uso permite que el desarrollador se centre principalmente en el manejo de elementos propios, es decir, a nivel de nodos en una lista o claves en un diccionario, en lugar de prestar más importancia a elementos de más bajo nivel, como punteros o buffers de memoria.

Uno de los aspectos claves a la hora de manejar de manera eficiente STL se basa en utilizar la herramienta adecuada para un problema concreto. Si no es así, entonces es fácil reducir enormemente el rendimiento de la aplicación debido a que no se utilizó, por ejemplo, la estructura de datos más adecuada.

5.2.3. Inconvenientes

El uso de STL también puede presentar ciertos inconvenientes; algunos de ellos directamente vinculados a su propia complejidad [27]. En este contexto, uno de los principales inconvenientes al utilizar STL es la **depuración** de programas, debido a aspectos como el uso extensivo de plantillas por parte de STL. Este planteamiento complica bastante la inclusión de puntos de ruptura y la depuración interactiva. Sin embargo, este problema no es tan relevante si se supone que STL ha sido extensamente probado y depurado, por lo que en teoría no sería necesario llegar a dicho nivel.

Por otra parte, visualizar de manera directa el contenido de los contenedores o estructuras de datos utilizadas puede resultar complejo. A veces, es muy complicado conocer exactamente a qué elemento está apuntando un iterador o simplemente ver todos los elementos de un vector.

La última desventaja es la **asignación de memoria**, debido a que se pretende que STL se utilice en cualquier entorno de cómputo de propósito general. En este contexto, es lógico suponer que dicho entorno tenga suficiente memoria y la penalización por asignar memoria no es crítica. Aunque esta situación es la más común a la hora de

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2271.html

plantear un desarrollo, en determinados casos esta suposición no se puede aplicar, como por ejemplo en el desarrollo de juegos para consolas de sobremesa. Note que este tipo de plataformas suelen tener restricciones de memoria, por lo que es crítico controlar este factor para obtener un buen rendimiento.

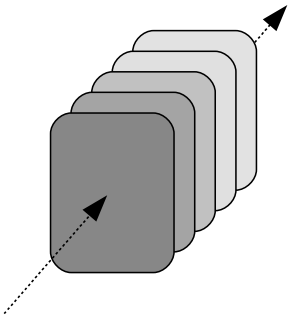


Figura 5.4: Los contenedores de secuencia mantienen un orden determinado a la hora de almacenar elementos, posibilitando optimizar el acceso y gestionando la consistencia caché.

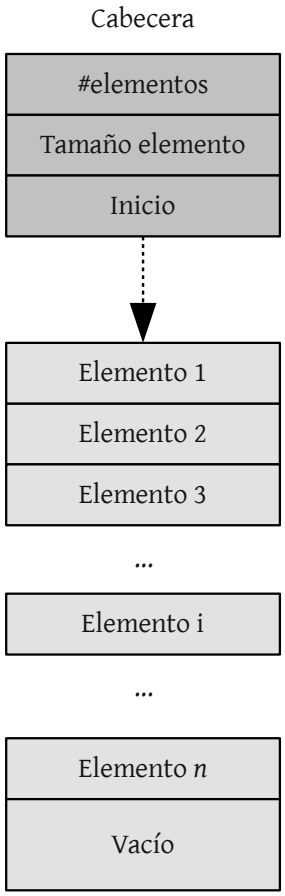


Figura 5.5: Implementación típica del contenedor vector con cabecera.

> En general, hacer uso de STL para desarrollar un videojuego es una de las mejores alternativas posibles. En el ámbito comercial, un gran número de juegos de ordenador y de consola han hecho uso de STL. Recuerde que siempre es posible personalizar algunos aspectos de STL, como la asignación de memoria, en función de las restricciones existentes.

Una posible solución a este inconveniente consiste en utilizar **asignadores** de memoria personalizados cuando se utilizan contenedores específicos, de manera que el desarrollador puede controlar cómo y cuándo se asigna memoria. Obviamente, esta solución implica que dicho desarrollador tenga un nivel de especialización considerable, ya que esta tarea no es trivial. En términos general, el uso de asignadores personalizados se lleva a cabo cuando el desarrollo de un videojuego se encuentra en un estado avanzado.

5.3. Secuencias

La principal característica de los contenedores de secuencia es que los elementos almacenados mantienen un **orden determinado**. La inserción y eliminación de elementos se puede realizar en cualquier posición debido a que los elementos residen en una secuencia concreta. A continuación se lleva a cabo una discusión de tres de los contenedores de secuencia más utilizados: el vector (*vector*), la cola de doble fin (*deque*) y la lista (*list*).

5.3.1. Vector

El vector es uno de los contenedores más simples y más utilizados de STL, ya que posibilita la inserción y eliminación de elementos en cualquier posición. Sin embargo, la complejidad computacional de dichas operaciones depende de la posición exacta en la que se inserta o elimina, respectivamente, el elemento en cuestión. Dicha complejidad determina el rendimiento de la operación y, por lo tanto, el rendimiento global del contenedor.

Un aspecto importante del vector es que proporciona iteradores bidireccionales, es decir, es posible acceder tanto al elemento posterior como al elemento anterior a partir de un determinado iterador. Así mismo, el vector permite el acceso directo sobre un determinado elemento, de manera similar al acceso en los arrays de C.

A diferencia de lo que ocurre con los arrays, un vector **no tiene límite** en cuanto al número de elementos que se pueden añadir. Al menos, mientras el sistema tenga memoria disponible. En caso de utilizar un array, el programador ha de comprobar continuamente el tamaño del mismo para asegurarse de que la insercción es factible, evitando así potenciales violaciones de segmento. En este contexto, el vector representa una solución a este tipo de problemas.

Los vectores proporcionan operaciones para añadir y eliminar elementos al final, insertar y eliminar elementos en cualquier posición y acceder a los mismos en tiempo constante a partir de un índice[2].

Implementación

Típicamente, el contenido de los vectores se almacena de **manera contigua** en un bloque de memoria, el cual suele contemplar el uso de espacio adicional para futuros elementos. Cada elemento de un vector se almacena utilizando un desplazamiento a la derecha, sin necesidad de hacer uso de punteros extra. De este modo, y partiendo de la base de que todos los elementos del vector ocupan el mismo tamaño debido a que son del mismo tipo, la localización de los mismos se calcula a partir del índice en la secuencia y del propio tamaño del elemento.

Los vectores utilizan una pequeña cabecera que contiene información general del contenedor, como el puntero al inicio del mismo, el número actual de elementos y el tamaño actual del bloque de memoria reservado (ver figura 5.5).

Rendimiento

En general, la inserción o eliminación de elementos al **final del vector** es muy eficiente. En este contexto, STL garantiza una complejidad constante en dichas operaciones, es decir, una complejidad $O(1)$. Desde otro punto de vista, la complejidad de estas operaciones es independiente del tamaño del vector. Además, la inserción es realmente eficiente y consiste en la simple copia del elemento a insertar.

La inserción o eliminación en **posiciones arbitrarias** requiere un mayor coste computacional debido a que es necesario *recolocar* los elementos a partir de la posición de inserción o eliminación en una posición más o una posición menos, respectivamente. La complejidad en estos casos es lineal, es decir, $O(n)$.

Es importante destacar el caso particular en el que la inserción de un elemento al final del vector implique la reasignación y copia del resto de elementos, debido a que el bloque de memoria inicialmente vinculado al vector esté lleno. Sin embargo, esta situación se puede evitar en bucles que requieran un rendimiento muy alto.

A pesar de la penalización a la hora de insertar y eliminar elementos en posiciones aleatorias, el vector podría ser la mejor opción si el programa a implementar no suele utilizar esta funcionalidad o si el número de elementos a manejar es bastante reducido.

Finalmente, el **recorrido transversal** de vectores es tan eficiente como el recorrido de elementos en un array. Para ello, simplemente hay que utilizar un iterador y alguna estructura de bucle, tal y como se muestra en el siguiente listado de código.

Listado 5.2: Recorrido básico de un vector

```
1 vector<int>::iterator it;
2
3 for (it = _vector.begin(); it != _vector.end(); ++it) {
4   int valor = *it;
5   // Utilizar valor...
6 }
```

Complejidad

La nomenclatura O se suele utilizar para acotar superiormente la complejidad de una determinada función o algoritmo. Los órdenes de complejidad más utilizados son $O(1)$ (complejidad constante), $O(logn)$ (complejidad logarítmica), $O(n)$ (complejidad lineal), $O(nlogn)$, $O(n^2)$ (complejidad cuadrática), $O(n^3)$ (complejidad cúbica), $O(n^x)$ (complejidad polinomial), $O(b^n)$ (complejidad exponencial).

[2] http://www.cplusplus.com/reference/stl/vector/

Respecto al **uso de memoria**, los vectores gestionan sus elementos en un gran bloque de memoria que permita almacenar los elementos actualmente contenidos y considere algún espacio extra para elementos futuros. Si el bloque inicial no puede albergar un nuevo elemento, entonces el vector reserva un bloque de memoria mayor, comúnmente con el doble de tamaño, copia el contenido del bloque inicial al nuevo y libera el bloque de memoria inicial. Este planteamiento evita que el vector tenga que reasignar memoria con frecuencia. En este contexto, es importante resaltar que un vector no garantiza la validez de un puntero o un iterador después de una inserción, ya que se podría dar esta situación. Por lo tanto, si es necesario acceder a un elemento en concreto, la solución ideal pasa por utilizar el índice junto con el operador **[]**.

Vectores y reserve()

La reserva de memoria explícita puede contribuir a mejorar el rendimiento de los vectores y evitar un alto número de operaciones de reserva.

Los vectores permiten la preasignación de un número de entradas con el objetivo de evitar la reasignación de memoria y la copia de elementos a un nuevo bloque. Para ello, se puede utilizar la operación **reserve** de vector que permite especificar la cantidad de memoria reservada para el vector y sus futuros elementos.

Listado 5.3: Reserva de memoria de vector

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  int main () {
7    vector<int> v;
8    v.reserve(4);
9    cout << "Capacidad inicial: " << v.capacity() << endl;
10
11   v.push_back(7); v.push_back(6);
12   v.push_back(4); v.push_back(6);
13   cout << "Capacidad actual: " << v.capacity() << endl;
14
15   v.push_back(4); // Provoca pasar de 4 a 8.
16   // Se puede evitar con v.reserve (8) al principio.
17   cout << "Capacidad actual: " << v.capacity() << endl;
18
19   return 0;
20 }
```

El caso contrario a esta situación está representado por vectores que contienen datos que se usan durante un cálculo en concreto pero que después se descartan. Si esta situación se repite de manera continuada, entonces se está asignando y liberando el vector constantemente. Una posible solución consiste en mantener el vector como estático y limpiar todos los elementos después de utilizarlos. Este planteamiento hace que el vector quede vacío y se llame al destructor de cada uno de los elementos previamente contenidos.

Finalmente, es posible aprovecharse del hecho de que los elementos de un vector se almacenan de manera contigua en memoria. Por ejemplo, sería posible pasar el contenido de un vector a una función que acepte un array, siempre y cuando dicha función no modifique su contenido [27], ya que el contenido del vector no estaría sincronizado.

¿Cuándo usar vectores?

Como regla general, los vectores se pueden utilizar extensivamente debido a su **buen rendimiento**. En concreto, los vectores deberían utilizarse en lugar de usar arrays estáticos, ya que proporcionan un enfoque muy flexible y no es necesario tener en cuenta el número de elementos del mismo para aplicar una operación de inserción, por ejemplo.

En el ámbito del desarrollo de videojuegos, algunos autores plantean utilizar el vector como contenedor base y plantear el uso de otros contenedores en su lugar si así lo requiere la aplicación a desarrollar. Mientras tanto, los vectores son la solución ideal para aspectos como los siguientes [27]:

- Lista de jugadores de un juego.
- Lista de vértices con geometría simplificada para la detección de colisiones.
- Lista de hijos de un nodo en un árbol, siempre que el tamaño de la misma no varíe mucho y no haya muchos hijos por nodo.
- Lista de animaciones asociada a una determinada entidad del juego.
- Lista de componentes de un juego, a nivel global.
- Lista de puntos que conforman la trayectoria de una cámara.
- Lista de puntos a utilizar por algoritmos de Inteligencia Artificial para el cálculo de rutas.

5.3.2. Deque

Deque proviene del término inglés *double-ended queue* y representa una **cola de doble fin**. Este tipo de contenedor es muy parecido al vector, ya que proporciona acceso directo a cualquier elemento y permite la inserción y eliminación en cualquier posición, aunque con distinto impacto en el rendimiento. La principal diferencia reside en que tanto la inserción como eliminación del primer y último elemento de la cola son muy rápidas. En concreto, tienen una complejidad constante, es decir, $O(1)$.

Deque y punteros

La cola de doble fin no garantiza que todos los elementos almacenados residan en direcciones contiguas de memoria. Por lo tanto, no es posible realizar un acceso seguro a los mismos mediante aritmética de punteros.

La colas de doble fin incluyen la funcionalidad básica de los vectores pero también considera operaciones para insertar y eliminar elementos, de manera explícita, al principio del contenedor[3].

Implementación

Este contenedor de secuencia, a diferencia de los vectores, mantiene varios bloques de memoria, en lugar de uno solo, de forma que se reservan nuevos bloques conforme el contenedor va creciendo en tamaño. Al contrario que ocurría con los vectores, no es necesario hacer copias de datos cuando se reserva un nuevo bloque de memoria, ya que los reservados anteriormente siguen siendo utilizados.

La cabecera de la cola de doble fin almacena una serie de punteros a cada uno de los bloques de memoria, por lo que su tamaño aumentará si se añaden nuevos bloques que contengan nuevos elementos.

[3] `http://www.cplusplus.com/reference/stl/deque/`

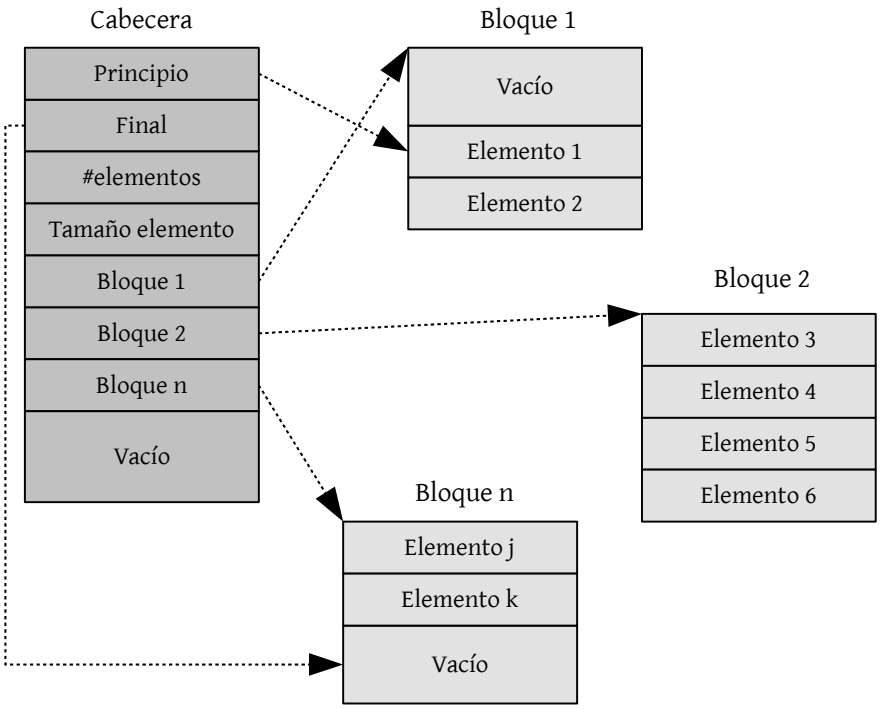


Figura 5.6: Implementación típica de *deque*.

Rendimiento

Al igual que ocurre con los vectores, la inserción y eliminación de elementos al final de la cola implica una complejidad constante, mientras que en una posición aleatoria en el centro del contenedor implica una complejidad lineal. Sin embargo, a diferencia del vector, la inserción y eliminación de elementos al principio es tan rápida como al final. La consecuencia directa es que las colas de doble fin son el candidato perfecto para estructuras FIFO (First In, First Out).

FIFO

Las estructuras First In, First Out se suelen utilizar en el ámbito del desarrollo de videojuegos para modelar colas de mensajes o colas con prioridad para atender peticiones asociadas a distintas tareas.

El hecho de manejar distintos bloques de memoria hace que el rendimiento se degrade mínimamente cuando se accede a un elemento que esté en otro bloque distinto. Note que dentro de un mismo bloque, los elementos se almacenan de manera secuencial al igual que en los vectores. Sin embargo, el recorrido transversal de una cola de doble fin es prácticamente igual de rápido que en el caso de un vector.

Respecto al **uso de memoria**, es importante resaltar que las asignaciones se producen de manera periódica durante el uso normal de la cola de doble fin. Por una parte, si se añaden nuevos elementos, entonces se reservan nuevos bloques memoria. Por otra parte, si se eliminan elementos, entonces se liberan otros bloques. Sin embargo, y aunque el número de elementos permanezca constante, es posible que se sigan asignando nuevos bloques si, por ejemplo, los elementos se añaden al final y se eliminan al principio. Desde un punto de vista abstracto, este contenedor se puede interpretar como una ventana deslizante en memoria, aunque siempre tenga el mismo tamaño.

Finalmente, y debido a la naturaleza dinámica de la cola de doble fin, no existe una función equivalente a la función *reserve* de vector.

¿Cuándo usar *deques*?

En general, la cola de doble fin puede no ser la mejor opción en entornos con restricciones de memoria, debido al mecanismo utilizado para reservar bloques de memoria. Desafortunadamente, este caso suele ser común en el ámbito del desarrollo de videojuegos. Si se maneja un número reducido de elementos, entonces el vector puede ser una alternativa más adecuada incluso aunque el rendimiento se degrade cuando se elimine el primer elemento de la estructura.

El uso de este contenedor no es, normalmente, tan amplio como el del vector. No obstante, existen situaciones en las que su utilización representa una opción muy buena, como por ejemplo la cola de mensajes a procesar en orden FIFO.

5.3.3. List

La lista es otro contenedor de secuencia que difiere de los dos anteriores. En primer lugar, la lista proporciona **iteradores bidireccionales**, es decir, iteradores que permiten navegar en los dos sentidos posibles: hacia adelante y hacia atrás. En segundo lugar, la lista no proporciona un acceso aleatorio a los elementos que contiene, como sí hacen tanto los vectores como las colas de doble fin. Por lo tanto, cualquier algoritmo que haga uso de este tipo de acceso no se puede aplicar directamente sobre listas.

La principal ventaja de la lista es el rendimiento y la conveniencia para determinadas operaciones, suponiendo que se dispone del iterador necesario para apuntar a una determinada posición.

La especificación de listas de STL ofrece una funcionalidad básica muy similar a la de cola de doble fin, pero también incluye funcionalidad relativa a aspectos más complejos como la fusión de listas o la búsqueda de elementos[4].

Algoritmos en listas

La propia naturaleza de la lista permite que se pueda utilizar con un buen rendimiento para implementar algoritmos o utilizar los ya existentes en la propia biblioteca.

Implementación

Este contenedor se implementa mediante una **lista doblemente enlazada de elementos**. Al contrario del planteamiento de los dos contenedores previamente discutidos, la memoria asociada a los elementos de la lista se reserva individualmente, de manera que cada nodo contiene un puntero al anterior elemento y otro al siguiente (ver figura 5.7).

La lista también tiene vinculada una cabecera con información relevante, destacando especialmente los punteros al primer y último elemento de la lista.

Rendimiento

La principal ventaja de una lista en términos de rendimiento es que la inserción y eliminación de elementos en cualquier posición se realiza en tiempo constante, es decir, $O(1)$. Sin embargo, para garantizar esta propiedad es necesario obtener un iterador a la posición deseada, operación que puede tener una complejidad no constante.

Una desventaja con respecto a los vectores y a las colas de doble fin es que el **recorrido transvesal** de las listas es mucho más lento, debido a que es necesario leer un puntero para acceder a cada nodo y, además, los nodos se encuentran en fragmentos de memoria no adyacentes. Desde un punto de vista general, el recorrido de elementos en una lista puede ser de hasta un orden de magnitud más lento que en vectores.

Entre listas...

Las operaciones que implican el movimiento de bloques de elementos en listas se realiza en tiempo constante, incluso cuando dichas operaciones se realizan utilizando más de un contenedor.

[4] `http://www.cplusplus.com/reference/stl/list/`

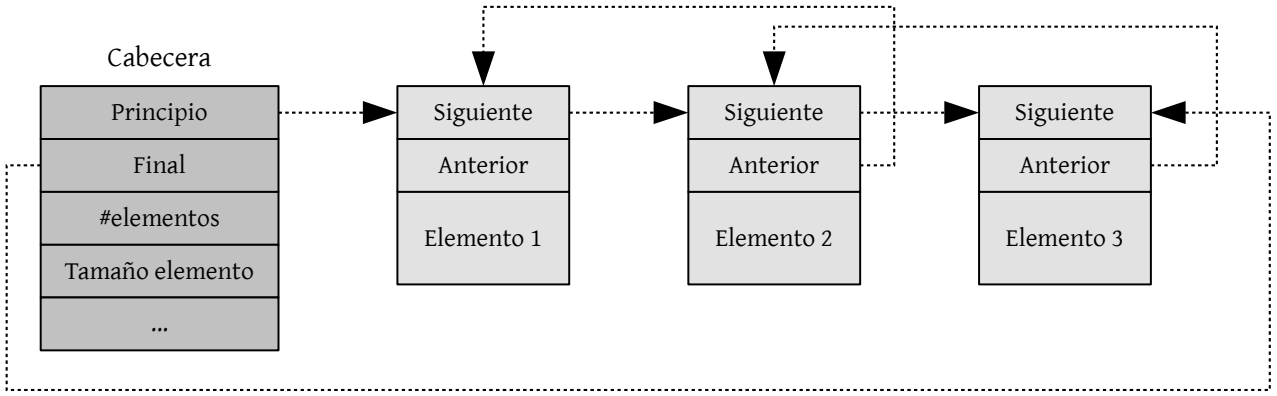


Figura 5.7: Implementación típica del contenedor lista.

Listado 5.4: Uso básico de listas con STL

```
1  #include <iostream>
2  #include <list>
3  #include <stdlib.h>
4  using namespace std;
5
6  class Clase {
7  public:
8    Clase (int id, int num_alumnos):
9      _id(id), _num_alumnos(num_alumnos) {}
10   int getId () const { return _id; }
11   int getNumAlumnos () const { return _num_alumnos; }
12
13   // Sobrecarga del operador para poder comparar clases.
14   bool operator< (const Clase &c) const {
15     return (_num_alumnos < c.getNumAlumnos());
16   }
17
18 private:
19   int _id;
20   int _num_alumnos;
21 };
22
23 void muestra_clases (list<Clase> lista);
24
25 int main () {
26   list<Clase> clases; // Lista de clases.
27   srand(time(NULL));
28
29   for (int i = 0; i < 7; ++i) // Inserción de clases.
30     clases.push_back(Clase(i, int(rand()  % 30 + 10)));
31
32   muestra_clases(clases);
33
34   // Se ordena la lista de clases.
35   // Usa la implementación del operador de sobrecarga.
36   clases.sort();
37   muestra_clases(clases);
38
39   return 0;
40 }
```

El planteamiento basado en manejar punteros hace que las listas sean más eficientes a la hora de, por ejemplo, reordenar sus elementos, ya que no es necesario realizar copias de datos. En su lugar, simplemente hay que modificar el valor de los punteros de acuerdo al resultado esperado o planteado por un determinado algoritmo. En este contexto, a mayor número de elementos en una lista, mayor beneficio en comparación con otro tipo de contenedores.

El anterior listado de código mostrará el siguiente resultado por la salida estándar:

```
0: 23 1: 28 2: 10 3: 17 4: 20 5: 22 6: 11
2: 10 6: 11 3: 17 4: 20 5: 22 0: 23 1: 28
```

es decir, mostrará la información relevante de los objetos de tipo *Clase* ordenados de acuerdo a la variable miembro *_num_alumnos*, ya que el valor de dicha variable se utiliza como criterio de ordenación. Este criterio se especifica de manera explícita al sobrecargar el operador $<$ (líneas 14-16). Internamente, la lista de STL hace uso de esta información para ordenar los elementos y, desde un punto de vista general, se basa en la sobrecarga de dicho operador para ordenar, entre otras funciones, tipos de datos definidos por el propio usuario.

Respecto al **uso en memoria**, el planteamiento de la lista reside en reservar pequeños bloques de memoria cuando se inserta un nuevo elemento en el contenedor. La principal ventaja de este enfoque es que no es necesario reasignar datos. Además, los punteros e iteradores a los elementos se preservan cuando se realizan inserciones y eliminaciones, posibilitando la implementación de distintas clases de algoritmos.

Evidentemente, la principal desventaja es que casi cualquier operación implica una nueva asignación de memoria. Este inconveniente se puede solventar utilizando asignadores de memoria personalizados, los cuales pueden usarse incluso para mejorar la penalización que implica tener los datos asociados a los nodos de la lista en distintas partes de la memoria.

Asignación de memoria

El desarrollo de juegos y la gestión de servidores de juego con altas cargas computacionales implica la implementación asignadores de memoria personalizados. Este planteamiento permite mejorar el rendimiento de alternativas clásicas basadas en el uso de *stacks* y *heaps*.

En el ámbito del desarrollo de videojuegos, especialmente en plataformas con restricciones de memoria, las listas pueden degradar el rendimiento debido a que cada nodo implica una cantidad extra, aunque pequeña, de memoria para llevar a cabo su gestión.

¿Cuándo usar listas?

Aunque las listas proporcionan un contenedor de uso general y se podrían utilizar como sustitutas de vectores o *deques*, su bajo rendimiento al iterar sobre elementos y la constante de reserva de memoria hacen que las listas no sean el candidato ideal para aplicaciones de alto rendimiento, como los videojuegos. Por lo tanto, las listas deberían considerarse como una tercera opción tras valores los otros dos contenedores previamente comentados.

A continuación se listan algunos de los posibles usos de listas en el ámbito del desarrollo de videojuegos:

- Lista de entidades del juego, suponiendo un alto número de inserciones y eliminaciones.
- Lista de mallas a renderizar en un *frame* en particular, suponiendo una ordenación en base a algún criterio como el material o el estado. En general, se puede evaluar el uso de listas cuando sea necesario aplicar algún criterio de ordenación.
- Lista dinámica de posibles objetivos a evaluar por un componente de Inteligencia Artificial, suponiendo un alto número de inserciones y eliminaciones.

Propiedad	Vector	Deque	List
I/E al final	$O(1)$	$O(1)$	$O(1)$
I/E al principio	$O(n)$	$O(1)$	$O(1)$
I/E en el medio	$O(n)$	$O(n)$	$O(1)$
Recorrido	Rápido	Rápido	Menos rápido
Asignación memoria	Raramente	Periódicamente	Con cada I/E
Acceso memoria sec.	Sí	Casi siempre	No
Invalidación iterador	Tras I/E	Tras I/E	Nunca

Cuadro 5.1: Resumen de las principales propiedades de los contenedores de secuencia previamente estudiados (I/E = inserción/eliminación).

5.4. Contenedores asociativos

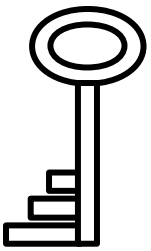


Figura 5.8: Los contenedores asociativos se basan en hacer uso de un elemento clave para acceder al contenido.

Mientras las secuencias se basan en la premisa de mantener posiciones relativas respecto a los elementos que contienen, los contenedores asociativos se desmarcan de dicho enfoque y están diseñados con el objetivo de **optimizar el acceso** a elementos concretos del contenedor de la manera más rápida posible.

En el caso de las secuencias, la búsqueda de elementos en el peor de los casos es de complejidad lineal, es decir, $O(n)$, ya que podría ser necesario iterar sobre todos los elementos del contenedor para dar con el elemento deseado. En algunos casos, es posible obtener una complejidad logarítmica, es decir, $O(logn)$, si los elementos de la secuencia están ordenados y se aplica una búsqueda binaria. Sin embargo, en general no es deseable mantener los elementos ordenados debido a que el rendimiento se verá degradado a la hora de insertar y eliminar elementos.

Por el contrario, los contenedores asociativos permiten obtener una complejidad logarítmica o incluso constante para encontrar un elemento concreto. Para ello, los elementos del contenedor asociativo se suelen indexar utilizando una clave que permite acceder al propio valor del elemento almacenado.

5.4.1. Set y multiset

En STL, un conjunto o *set* sigue la misma filosofía que un conjunto matemático, es decir, sirve como almacén para una serie de elementos. En esencia, un elemento se encuentra en un conjunto o no se encuentra en el mismo.

La inserción de objetos se realiza con la operación **insert**, la eliminación con **erase** y la búsqueda mediante **find**[5]. Esta operación tiene una complejidad logarítmica.

Un conjunto sólo tiene una instancia de cada objeto, de manera que insertar varias veces el mismo objeto produce el mismo resultado que insertarlo una única vez, es decir, no modifica el conjunto. De este modo, un conjunto es el candidato ideal para introducir un número de objetos y obviar los que sean redundantes. Este enfoque es mucho más eficiente que, por ejemplo, mantener una lista y hacer una búsqueda de elementos duplicados, ya que el algoritmo implicaría una complejidad cuadrática, es decir, $O(n^2)$.

[5] `http://www.cplusplus.com/reference/stl/set/`

STL también soporte el contenedor **multiset** o multi-conjunto[6], que sí permite mantener varias copias de un mismo objeto siempre y cuando se inserte en más de una ocasión. Este contenedor también permite acceder al número de veces que una determinada instancia se encuentra almacenada mediante la operación **count**.

Claves en sets

El contenedor set tiene como característica principal que los elementos almacenados en el mismo actúan como las propias claves.

A continuación se muestra un ejemplo de uso básico de los conjuntos. El aspecto más relevante del mismo es la declaración del conjunto en la línea (17), de manera que posibilite almacenar elementos enteros y, al mismo tiempo, haga uso de la definición de la clase *ValorAbsMenos* para definir un criterio de inclusión de elementos en el conjunto. Desde un punto de vista general, los conjuntos posibilitan la inclusión de criterios para incluir o no elementos en un conjunto. En el ejemplo propuesto, este criterio se basa en que la distancia entre los enteros del conjunto no ha de superar un umbral (DISTANCIA) para poder pertenecer al mismo. Note cómo la comparación se realiza en las líneas (9-11) con el operador $<$, es decir, mediante la función *menor*. Este enfoque es bastante común en STL, en lugar de utilizar el operador de igualdad.

Listado 5.5: Uso básico de conjuntos con STL

```
1  #include <iostream>
2  #include <set>
3  #include <algorithm>
4  using namespace std;
5
6  #define DISTANCIA 5
7
8  struct ValorAbsMenos {
9    bool operator() (const int& v1, const int& v2) const {
10     return (abs(v1 - v2) < DISTANCIA);
11   }
12 };
13
14 void recorrer (set<int, ValorAbsMenos> valores);
15
16 int main () {
17   set<int, ValorAbsMenos> valores;
18
19   valores.insert(5);  valores.insert(9);
20   valores.insert(3);  valores.insert(7);
21
22   recorrer(valores);
23
24   return 0;
25 }
```

La salida al ejecutar el listado de código anterior se expone a continuación:

```
7 9 5
```

es decir, el elemento 3 no se encuentra en el conjunto debido a que su distancia con el elemento 9 es mayor a la definida en el umbral (5).

Finalmente, *set* y *multiset* son **contenedores asociativos ordenados**, debido a que los elementos se almacenan utilizando un cierto orden, el cual se basa en una función de comparación como la discutida anteriormente. Este enfoque permite encontrar elementos rápidamente e iterar sobre los mismos en un determinado orden.

[6] http://www.cplusplus.com/reference/stl/multiset/

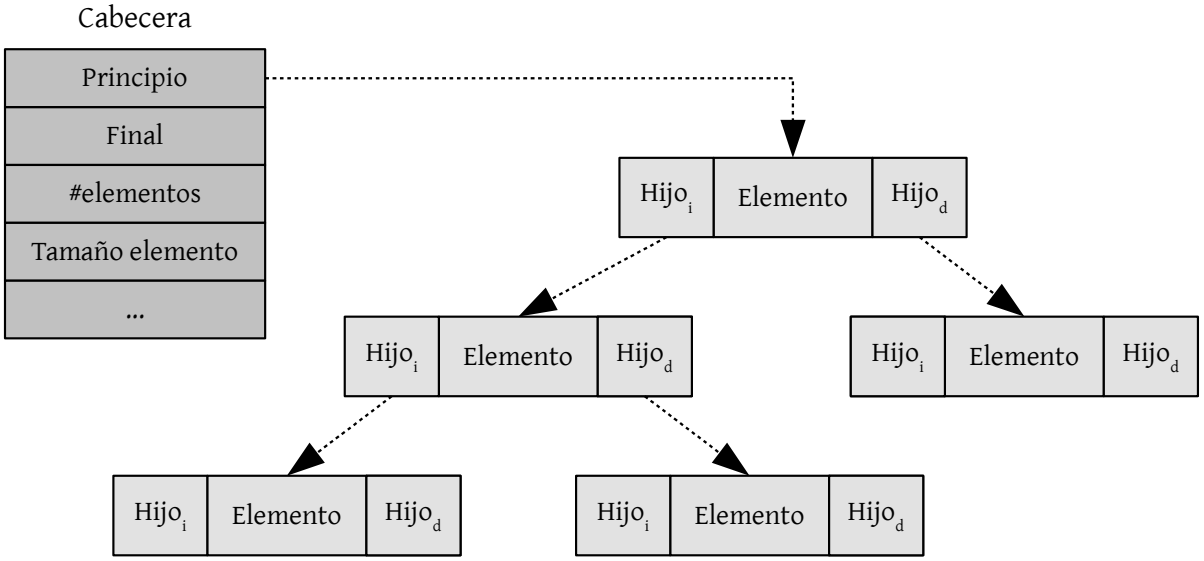


Figura 5.9: Implementación típica del contenedor set.

Implementación

En términos generales, los conjuntos se implementan utilizando un **árbol binario balanceado** (ver figura 5.9) para garantizar que la complejidad de las búsquedas en el peor caso posible sea logarítmica, es decir, $O(logn)$. Cada elemento, al igual que ocurre con la lista, está vinculado a un nodo que enlaza con otros manteniendo una estructura de árbol. Dicho árbol se ordena con la función de comparación especificada en la plantilla. Aunque este enfoque es el más común, es posible encontrar alguna variación en función de la implementación de STL utilizada.

Rendimiento

La principal baza de los conjuntos es su gran rendimiento en operaciones relativas a la **búsqueda de elementos**, garantizando una complejidad logarítmica en el peor de los casos. Sin embargo, es importante tener en cuenta el tiempo invertido en la comparación de elementos, ya que ésta no es trivial en caso de manejar estructuras de datos más complejas, como por ejemplo las matrices. Así mismo, la potencia de los conjuntos se demuestra especialmente cuando el número de elementos del contenedor comienza a crecer.

Comparando elementos

Aunque los conjuntos permiten realizar búsquedas eficientes, es importante tener en cuenta la complejidad asociada a la propia comparación de elementos, especialmente si se utilizan estructuras de datos complejas.

Una alternativa posible para buscar elementos de manera eficiente puede ser un vector, planteando para ello una ordenación de elementos seguida de una búsqueda binaria. Este planteamiento puede resultar eficiente si el número de búsquedas es muy reducido y el número de accesos es elevado.

La inserción de elementos en un árbol balanceado tiene una complejidad logarítmica, pero puede implicar la reordenación de los mismos para mantener balanceada la estructura. Por otra parte, el recorrido de los elementos de un conjunto es muy similar al de listas, es decir, se basa en el uso de manejo de enlaces a los siguientes nodos de la estructura.

Respecto al **uso de memoria**, la implementación basada en árboles binarios balanceados puede no ser la mejor opción ya que la inserción de elementos implica la asignación de un nuevo nodo, mientras que la eliminación implica la liberación del mismo.

¿Cuándo usar sets y multisets?

En general, este tipo de contenedores asociativos están ideados para situaciones específicas, como por ejemplo el hecho de manejar conjuntos con elementos no redundantes o cuando el número de búsquedas es relevante para obtener un rendimiento aceptable. En el ámbito del desarrollo de videojuegos, este tipo de contenedores se pueden utilizar para mantener un conjunto de objetos de manera que sea posible cargar de manera automática actualizaciones de los mismos, permitiendo su sobreescritura en función de determinados criterios.

5.4.2. Map y multimap

En STL, el contenedor *map* se puede interpretar como una extensión de los conjuntos con el principal objetivo de optimizar aún más la búsqueda de elementos. Para ello, este contenedor distingue entre **clave** y **valor** para cada elemento almacenado en el mismo. Este planteamiento difiere de los conjuntos, ya que estos hacen uso del propio elemento como clave para ordenar y buscar. En esencia, un *map* es una secuencia de pares <clave, valor>, proporcionando una recuperación rápida del valor a partir de dicha clave. Así mismo, un *map* se puede entender como un tipo especial de array que permite la indexación de elementos mediante cualquier tipo de datos. De hecho, el contenedor permite el uso del operador **[]**.

El **multimap** permite manejar distintas instancias de la misma clave, mientras que el *map* no. Es importante resaltar que no existe ninguna restricción respecto al valor, es decir, es posible tener un mismo objeto como valor asociado a distintas claves.

La principal aplicación de un *map* es manejar estructuras de tipo diccionario que posibiliten el acceso inmediato a un determinado valor dada una clave. Por ejemplo, sería posible asociar un identificador único a cada una de las entidades de un juego y recuperar los punteros asociados a las propias entidades cuando así sea necesario. De este modo, es posible utilizar sólo la memoria necesaria cuando se añaden nuevas entidades.

Tanto el *map*[7] como el *multimap*[8] son contenedores asociativos ordenados, por lo que es posible recorrer su contenido en el orden en el que estén almacenados. Sin embargo, este orden no es el orden de inserción inicial, como ocurre con los conjuntos, sino que viene determinado por aplicar una función de comparación.

Implementación

La implementación típica de este tipo de contenedores es idéntica a la de los conjuntos, es decir, mediante un **árbol binario balanceado**. Sin embargo, la diferencia reside en que las claves para acceder y ordenar los elementos son objetos distintos a los propios elementos.

[7] `http://www.cplusplus.com/reference/stl/map/`
[8] `http://www.cplusplus.com/reference/stl/multimap/`

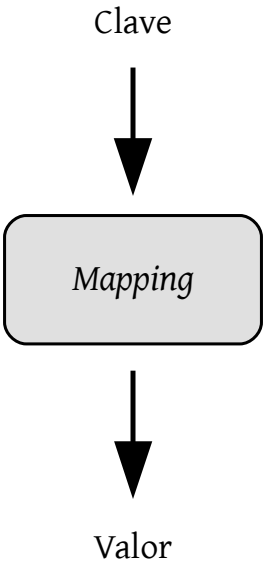


Figura 5.10: El proceso de utilizar una clave en un map para obtener un valor se suele denominar comúnmente *mapping*.

Rendimiento

El rendimiento de estos contenedores es prácticamente idéntico al de los conjuntos, salvo por el hecho de que las comparaciones se realizan a nivel de clave. De este modo, es posible obtener cierta ventaja de **manejar claves simples** sobre una gran cantidad de elementos complejos, mejorando el rendimiento de la aplicación. Sin embargo, nunca hay que olvidar que si se manejan claves más complejas como cadenas o tipos de datos definidos por el usuario, el rendimiento puede verse afectado negativamente.

El operador []

El operador [] se puede utilizar para acceder, escribir o actualizar información sobre algunos contenedores. Sin embargo, es importante considerar el impacto en el rendimiento al utilizar dicho operador, el cual vendrá determinado por el contenedor usado.

Por otra parte, el uso del operador [] no tiene el mismo rendimiento que en vectores, ya que implica realizar la búsqueda del elemento a partir de la clave y, por lo tanto, no sería de un orden de complejidad constante sino logarítmico. Al usar este operador, también hay que tener en cuenta que si se escribe sobre un elemento que no existe, entonces éste se añade al contenedor. Sin embargo, si se intenta leer un elemento que no existe, entonces el resultado es que el elemento por defecto se añade para la clave en concreto y, al mismo tiempo, se devuelve dicho valor. El siguiente listado de código muestra un ejemplo.

Como se puede apreciar, el listado de código muestra características propias de STL para manejar los elementos del contenedor:

- En la línea 7 se hace uso de *pair* para declarar una pareja de valores: i) un iterador al contenedor definido en la línea 6 y ii) un valor booleano. Esta estructura se utilizará para recoger el valor de retorno de una inserción (línea 14).
- Las líneas 10–11 muestran inserciones de elementos.
- En la línea 15 se recoge el valor de retorno al intentar insertar un elemento con una clave ya existente. Note cómo se accede al iterador en la línea 17 para obtener el valor previamente almacenado en la entrada con clave 2.
- La línea 20 muestra un ejemplo de inserción con el operador [], ya que la clave 3 no se había utilizado previamente.
- La línea 23 ejemplifica la inserción de un elemento por defecto. Al tratarse de cadenas de texto, el valor por defecto es la cadena vacía.

Listado 5.6: Uso básico de map con STL

```
1  #include <iostream>
2  #include <map>
3
4  using namespace std;
5
6  int main () {
7    map<int, string> jugadores;
8    pair<map<int, string>::iterator, bool> ret;
9
10   // Insertar elementos.
11   jugadores.insert(pair<int, string>(1, "Luis"));
12   jugadores.insert(pair<int, string>(2, "Sergio"));
13
14   // Comprobando elementos ya insertados...
15   ret = jugadores.insert(pair<int, string>(2, "David"));
16
17   if (ret.second == false) {
18     cout << "El elemento 2 ya existe ";
19     cout << "con un valor de " << ret.first->second << endl;
20   }
21
```

Propiedad	Set	Map
I/E elementos	$O(logn)$	$O(logn)$
Búsqueda	$O(logn)$	$O(logn)$
Recorrido	Más lento que lista	Más lento que lista
Asignación memoria	Con cada I/E	Con cada I/E
Acceso memoria sec.	No	No
Invalidación iterador	Nunca	Nunca

Cuadro 5.2: Resumen de las principales propiedades de los contenedores asociativos previamente estudiados (I/E = inserción/eliminación).

```
22    jugadores[3] = "Alfredo"; // Inserción con []...
23
24    // Caso excepcional; se añade valor por defecto...
25    const string& j_aux = jugadores[4]; // jugadores[4] = ”
26
27    return 0;
28  }
```

Usando referencias

Recuerde consultar información sobre las operaciones de cada uno de los contenedores estudiados para conocer exactamente su signatura y cómo invocarlas de manera eficiente.

Ante esta situación, resulta deseable hacer uso de la operación **find** para determinar si un elemento pertenece o no al contenedor, accediendo al mismo con el iterador devuelto por dicha operación. Así mismo, la inserción de elementos es más eficiente con *insert* en lugar de con el operador [], debido a que este último implica un mayor número de copias del elemento a insertar. Por el contrario, [] es ligeramente más eficiente a la hora de actualizar los contenidos del contenedor en lugar de *insert*.

Respecto al **uso de memoria**, los mapas tienen un comportamiento idéntico al de los conjuntos, por lo que se puede suponer los mismos criterios de aplicación.

¿Cuándo usar maps y multimaps?

En general, estos contenedores se utilizan como diccionarios de rápido acceso que permiten indexar elementos a partir de manejadores o identificadores únicos. Sin embargo, también es posible utilizar claves más complejas, como por ejemplo cadenas. Algunas de las principales aplicaciones en el desarrollo de videojuegos son las siguientes:

- Mantenimiento de un diccionario con identificadores únicos para mapear las distintas entidades del juego. De este modo, es posible evitar el uso de punteros a dichas entidades.
- Mecanismo de traducción entre elementos del juego, como por ejemplo entre identificadores numéricos y cadenas de texto vinculadas a los nombres de los personajes.

5.5. Adaptadores de secuencia

STL proporciona el concepto de adaptadores para proporcionar una funcionalidad más restringida sobre un contenedor existente sin la necesidad de que el propio desarrollador tenga que especificarla. Este enfoque se basa en que, por ejemplo, los contenederos de secuencia existentes tienen la flexibilidad necesaria para comportarse como otras estructuras de datos bien conocidas, como por ejemplo las **pilas** o las **colas**.

LIFO

La pila está diseña para operar en un contexto LIFO (Last In, First Out) (*last-in first-out*), es decir, el elemento que se apiló más recientemente será el primero en salir de la pila.

Un pila es una estructura de datos que solamente permite añadir y eliminar elementos por un extremo, la cima. En este contexto, cualquier conteneder de secuencia discutido anteriormente se puede utilizar para proporcionar dicha funcionalidad. En el caso particular de STL, es posible manejar pilas e incluso especificar la implementación subyacente que se utilizará, especificando de manera explícita cuál será el contenedor de secuencia usado.

Aunque sería perfectamente posible delegar en el programador el manejo del contenedor de secuencia para que se comporte como, por ejemplo, una pila, en la práctica existen dos razones importantes para la definición de adaptadores:

1. La declaración explícita de un adaptador, como una pila o *stack*, hace que sea el código sea mucho más claro y legible en términos de funcionalidad. Por ejemplo, declarar una pila proporciona más información que declarar un vector que se comporte como una pila. Esta aproximación facilita la interoperabilidad con otros programadores.
2. El compilador tiene más información sobre la estructura de datos y, por lo tanto, puede contribuir a la detección de errores con más eficacia. Si se utiliza un vector para modelar una pila, es posible utilizar alguna operación que no pertenezca a la interfaz de la pila.

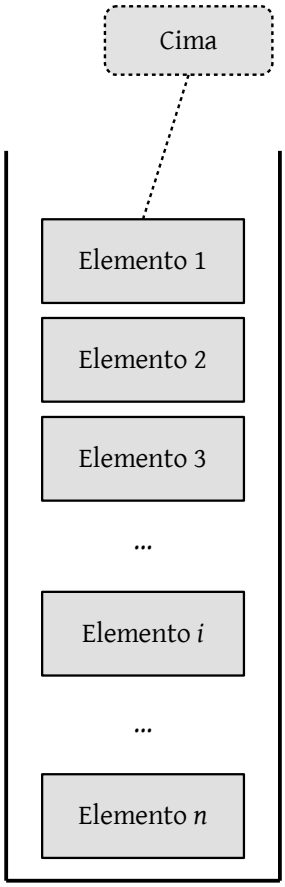


Figura 5.11: Visión abstracta de una pila o *stack*. Los elementos solamente se añaden o eliminan por un extremo: la cima.

5.5.1. Stack

El adaptador más sencillo en STL es la pila o *stack*[9]. Las principales operaciones sobre una pila son la inserción y eliminación de elementos por uno de sus extremos: la cima. En la literatura, estas operaciones se conocen como **push** y **pop**, respectivamente.

La pila es más restrictiva en términos funcionales que los distintos contenedores de secuencia previamente estudiados y, por lo tanto, se puede implementar con cualquiera de ellos. Obviamente, el rendimiento de las distintas versiones vendrá determinado por el contenedor elegido. Por defecto, la pila se implementa utilizando una cola de doble fin.

El siguiente listado de código muestra cómo hacer uso de algunas de las operaciones más relevantes de la pila para invertir el contenido de un vector.

Listado 5.7: Inversión del contenido de un vector con una pila

```
1 #include <iostream>
2 #include <stack>
3 #include <vector>
4 using namespace std;
5
6 int main () {
7   vector<int> fichas;
```

[9] http://www.cplusplus.com/reference/stl/stack/

```
 8   vector<int>::iterator it;
 9   stack<int> pila;
10
11   for (int i = 0; i < 10; i++) // Rellenar el vector
12     fichas.push_back(i);
13
14   for (it = fichas.begin(); it != fichas.end(); ++it)
15     pila.push(*it); // Apilar elementos para invertir
16
17   fichas.clear(); // Limpiar el vector
18
19   while (!pila.empty()) { // Rellenar el vector
20     fichas.push_back(pila.top());
21     pila.pop();
22   }
23
24   return 0;
25 }
```

5.5.2. Queue

Al igual que ocurre con la pila, la cola o *queue*[10] es otra estructura de datos de uso muy común que está representada en STL mediante un adaptador de secuencia. Básicamente, la cola mantiene una interfaz que permite la inserción de elementos al final y la extracción de elementos sólo por el principio. En este contexto, no es posible acceder, insertar y eliminar elementos que estén en otra posición.

Por defecto, la cola utiliza la implementación de la cola de doble fin, siendo posible utilizar la implementación de la lista. Sin embargo, no es posible utilizar el vector debido a que este contenedor no proporciona la operación *push_front*, requerida por el propio adaptador. De cualquier modo, el rendimiento de un vector para eliminar elementos al principio no es particularmente bueno, ya que tiene una complejidad lineal.

5.5.3. Cola de prioridad

STL también proporciona el adaptador cola de prioridad o *priority queue*[11] como caso especial de cola previamente discutido. La diferencia entre los dos adaptadores reside en que el elemento listo para ser eliminado de la cola es aquél que tiene una mayor prioridad, no el elemento que se añadió en primer lugar.

Así mismo, la cola con prioridad incluye cierta funcionalidad específica, a diferencia del resto de adaptadores estudiados. Básicamente, cuando un elemento se añade a la cola, entonces éste se ordena de acuerdo a una función de prioridad.

Por defecto, la cola con prioridad se apoya en la implementación de vector, pero es posible utilizarla también con *deque*. Sin embargo, no se puede utilizar una lista debido a que la cola con prioridad requiere un acceso aleatorio para insertar eficientemente elementos ordenados.

La comparación de elementos sigue el mismo esquema que el estudiado en la sección 5.4.1 y que permitía definir el criterio de inclusión de elementos en un conjunto, el cual estaba basado en el uso del operador *menor que*.

[10] http://www.cplusplus.com/reference/stl/queue/

[11] http://www.cplusplus.com/reference/stl/priority_queue/

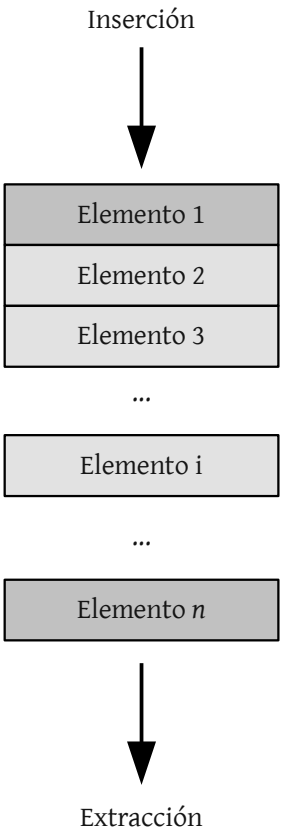


Figura 5.12: Visión abstracta de una cola o *queue*. Los elementos solamente se añaden por el final y se eliminan por el principio.

La cola de prioridad se puede utilizar para múltiples aspectos en el desarrollo de videojuegos, como por ejemplo la posibilidad de mantener una estructura ordenada por prioridad con aquellos objetos que están más cercanos a la posición de la cámara virtual.

Capítulo 6

Gestión de Recursos

David Vallejo Fernández

En este capítulo se cubren aspectos esenciales a la hora de afrontar el diseño de un juego. En particular, se discutirán las arquitecturas típicas del bucle de juego, haciendo especial hincapié en un esquema basado en la gestión de los estados de un juego. Como caso de estudio concreto, en este capítulo se propone una posible implementación del bucle principal mediante este esquema haciendo uso de las bibliotecas que Ogre3D proporciona.

Así mismo, este capítulo discute la gestión de recursos y, en concreto, la gestión de sonido y de efectos especiales. La gestión de recursos es especialmente importante en el ámbito del desarrollo de videojuegos, ya que el rendimiento de un juego depende en gran medida de la eficiencia del subsistema de gestión de recursos.

Finalmente, la gestión del sistema de archivos, junto con aspectos básicos de entrada/salida, también se estudia en este capítulo. Además, se plantea el diseño e implementación de un importador de datos que se puede utilizar para integrar información multimedia a nivel de código fuente.

6.1. El bucle de juego

6.1.1. El bucle de renderizado

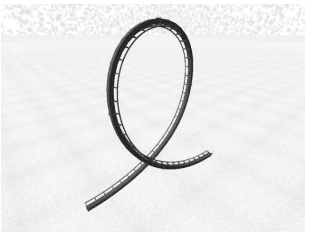


Figura 6.1: El bucle de juego representa la estructura de control principal de cualquier juego y gobierna su funcionamiento y la transición entre los distintos estados del mismo.

Hace años, cuando aún el **desarrollo de videojuegos 2D** era el estándar en la industria, uno de los principales objetivos de diseño de los juegos era minimizar el número de píxeles a dibujar por el *pipeline* de renderizado con el objetivo de maximizar la tasa de *fps* del juego. Evidentemente, si en cada una de las iteraciones del bucle de renderizado el número de píxeles que cambia es mínimo, el juego *correrá* a una mayor velocidad.

Esta técnica es en realidad muy parecida a la que se plantea en el desarrollo de interfaces gráficas de usuario (GUI (Graphical User Interface)), donde gran parte de las mismas es estática y sólo se producen cambios, generalmente, en algunas partes bien definidas. Este planteamiento, similar al utilizado en el desarrollo de videojuegos 2D antiguos, está basado en *redibujar* únicamente aquellas partes de la pantalla cuyo contenido cambia.

Rectangle invalidation

La técnica basada en *redibujar* únicamente aquellas partes de la pantalla cuyo contenido cambia realmente se suele denominar *rectangle invalidation*.

En el **desarrollo de videojuegos 3D**, aunque manteniendo la idea de dibujar el mínimo número de primitivas necesarias en cada iteración del bucle de renderizado, la filosofía es radicalmente distinta. En general, al mismo tiempo que la cámara se mueve en el espacio tridimensional, el contenido audiovisual cambia continuamente, por lo que no es viable aplicar técnicas tan simples como la mencionada anteriormente.

La consecuencia directa de este esquema es la necesidad de un bucle de renderizado que muestre los distintas imágenes o frames percibidas por la cámara virtual con una velocidad lo suficientemente elevada para transmitir una sensación de realidad.

El siguiente listado de código muestra la **estructura general** de un bucle de renderizado.

Listado 6.1: Esquema general de un bucle de renderizado.

```
1  while (true) {
2    // Actualizar la cámara,
3    // normalmente de acuerdo a un camino prefijado.
4    update_camera ();
5
6    // Actualizar la posición, orientación y
7    // resto de estado de las entidades del juego.
8    update_scene_entities ();
9
10   // Renderizar un frame en el buffer trasero.
11   render_scene ();
12
13   // Intercambiar el contenido del buffer trasero
14   // con el que se utilizará para actualizar el
15   // dispositivo de visualización.
16   swap_buffers ();
17 }
```

6.1.2. Visión general del bucle de juego

Keep it simple, Stupid!

La filosofía KISS (Keep it simple, Stupid!) se adapta perfectamente al planteamiento del bucle de juego, en el que idealmente se implementa un enfoque sencillo, flexible y escalable para gestionar los distintos estados de un juego.

Como ya se introdujo en la sección 1.2, en un motor de juegos existe una gran variedad de subsistemas o componentes con distintas necesidades. Algunos de los más importantes son el motor de renderizado, el sistema de detección y gestión de colisiones, el subsistema de juego o el subsistema de soporte a la Inteligencia Artificial.

La mayoría de estos componentes han de actualizarse periódicamente mientras el juego se encuentra en ejecución. Por ejemplo, el sistema de animación, de manera sincronizada con respecto al motor de renderizado, ha de actualizarse con una frecuencia de 30 ó 60 Hz con el objetivo de obtener una tasa de *frames* por segundo lo suficientemente elevada para garantizar una sensación de realismo adecuada. Sin embargo, no es necesario mantener este nivel de exigencia para otros componentes, como por ejemplo el de Inteligencia Artificial.

De cualquier modo, es necesario un planteamiento que permita actualizar el estado de cada uno de los subsistemas y que considere las restricciones temporales de los mismos. Típicamente, este planteamiento se suele abordar mediante el **bucle de juego**, cuya principal responsabilidad consiste en actualizar el estado de los distintos componentes del motor tanto desde el punto de vista interno (ej. coordinación entre subsistemas) como desde el punto de vista externo (ej. tratamiento de eventos de teclado o ratón).

Listado 6.2: Esquema general del bucle de juego.

```
1  // Pseudocódigo de un juego tipo "Pong".
2  int main (int argc, char* argv[]) {
3    init_game();              // Inicialización del juego.
4
5    // Bucle del juego.
6    while (1) {
7      capture_events();       // Capturar eventos externos.
8
9      if (exitKeyPressed())   // Salida.
10       break;
11
12     move_paddles();         // Actualizar palas.
13     move_ball();            // Actualizar bola.
14     collision_detection(); // Tratamiento de colisiones.
15
16     // ¿Anotó algún jugador?
17     if (ballReachedBorder(LEFT_PLAYER)) {
18       score(RIGHT_PLAYER);
19       reset_ball();
20     }
21     if (ballReachedBorder(RIGHT_PLAYER)) {
22       score(LEFT_PLAYER);
23       reset_ball();
24     }
25
26     render();               // Renderizado.
27   }
28 }
```

Antes de discutir algunas de las arquitecturas más utilizadas para modelar el bucle de juego, resulta interesante estudiar el anterior listado de código, el cual muestra una manera muy simple de gestionar el bucle de juego a través de una sencilla estructura de control iterativa. Evidentemente, la complejidad actual de los videojuegos comerciales requiere un esquema que sea más general y escalable. Sin embargo, es muy importante mantener la simplicidad del mismo para garantizar su mantenibilidad.

Figura 6.2: El hardware de Atari Pong estaba especialmente pensado para el manejo de las dos palas que forman el juego.

6.1.3. Arquitecturas típicas del bucle de juego

La arquitectura del bucle de juego se puede implementar de diferentes formas mediante distintos planteamientos. Sin embargo, la mayoría de ellos tienen en común el uso de uno o varios **bucles de control** que gobiernan la actualización e interacción con los distintos componentes del motor de juegos. En esta sección se realiza un breve recorrido por las alternativas más populares, resaltando especialmente un planteamiento basado en la gestión de los distintos estados por los que puede atravesar un juego. Esta última alternativa se discutirá con un caso de estudio detallado que hace uso de Ogre.

Tratamiento de mensajes en Windows

En las plataformas Windows™, los juegos han de atender los mensajes recibidos por el propio sistema operativo y dar soporte a los distintos componentes del propio motor de juego. Típicamente, en estas plataformas se implementan los denominados ***message pumps*** [42], como responsables del tratamiento de este tipo de mensajes.

Desde un punto de vista general, el planteamiento de este esquema consiste en atender los mensajes del propio sistema operativo cuando llegan, interactuando con el motor de juegos cuando no existan mensajes del sistema operativo por procesar. En ese caso se ejecuta una iteración del bucle de juego y se repite el mismo proceso.

La principal consecuencia de este enfoque es que los mensajes del sistema operativo tienen prioridad con respecto a aspectos críticos como el bucle de renderizado. Por ejemplo, si la propia ventana en la que se está ejecutando el juego se arrastra o su tamaño cambia, entonces el juego se *congelará* a la espera de finalizar el tratamiento de eventos recibidos por el propio sistema operativo.

Esquemas basados en retrollamadas

El concepto de retrollamada o ***callback***, introducido de manera implícita en la discusión del patrón MVC de la sección 4.3.4, consiste en asociar una porción de código para atender un determinado evento o situación. Este concepto se puede asociar a una función en particular o a un objeto. En este último caso, dicho objeto se denominará *callback object*, término muy usado en el desarrollo de interfaces gráficas de usuario y en el área de los sistemas distribuidos.

A continuación se muestra un ejemplo de uso de funciones de retrollamada planteado en la biblioteca GLUT, la cual está estrechamente ligada a la biblioteca GL, utilizada para tratar de manera simple eventos básicos.

Listado 6.3: Ejemplo de uso de retrollamadas con OpenGL.

```
1  #include <GL/glut.h>
2  #include <GL/glu.h>
3  #include <GL/gl.h>
4
5  // Se omite parte del código fuente...
6
7  void update (unsigned char key, int x, int y) {
8    Rearthyear += 0.2;
9    Rearthday += 5.8;
10   glutPostRedisplay();
11 }
12
13 int main (int argc, char** argv) {
14     glutInit(&argc, argv);
15
16     glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
```

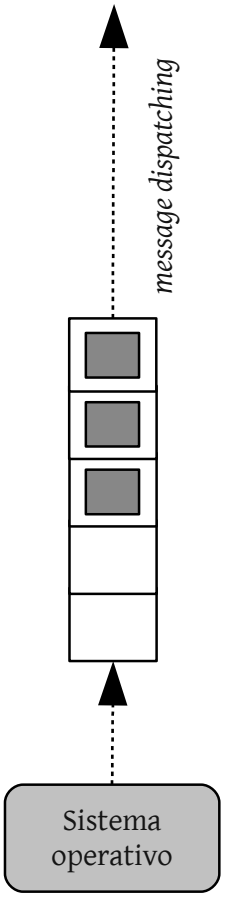


Figura 6.3: Esquema gráfico de una arquitectura basada en *message pumps*.

```
17      glutInitWindowSize(640, 480);
18      glutCreateWindow("Session #04 - Solar System");
19
20      // Definición de las funciones de retrollamada.
21      glutDisplayFunc(display);
22      glutReshapeFunc(resize);
23      // Eg. update se ejecutará cuando el sistema
24      // capture un evento de teclado.
25      // Signatura de glutKeyboardFunc:
26      // void glutKeyboardFunc(void (*func)
27      // (unsigned char key, int x, int y));
28      glutKeyboardFunc(update);
29
30      glutMainLoop();
31
32      return 0;
33  }
```


Desde un punto de vista abstracto, las funciones de retrollamada se suelen utilizar como mecanismo para *rellenar* el código fuente necesario para tratar un determinado tipo de evento. Este esquema está directamente ligado al concepto de ***framework***, entendido como una aplicación construida parcialmente y que el desarrollador ha de *completar* para proporcionar una funcionalidad específica.

Figura 6.4: Un *framework* proporciona al desarrollador una serie de herramientas para solucionar problemas dependientes de un dominio. El propio desarrollador tendrá que utilizar las herramientas disponibles y ajustarlas para proporcionar una buena solución.

Algunos autores [42] definen a Ogre3D como un *framework* que envuelve a una biblioteca que proporciona, principalmente, funcionalidad asociada a un motor de renderizado. Sin embargo, ya se ha discutido cómo Ogre3D proporciona una gran cantidad de herramientas para el desarrollo de aplicaciones gráficas interactivas en 3D.

Las instancias de la clase **Ogre::FrameListener** son un ejemplo representativo de objetos de retrollamada, con el objetivo de que el desarrollador pueda decidir las acciones que se han de ejecutar antes y después de que se produzca una iteración en el bucle de renderizado. Dicha funcionalidad está representada por las funciones virtuales *frameStarted()* y *frameEnded()*, respectivamente. En el Módulo 2, *Programación Gráfica*, se discute en profundidad un ejemplo de uso de esta clase.

Tratamiento de eventos

En el ámbito de los juegos, un **evento** representa un cambio en el estado del propio juego o en el entorno. Un ejemplo muy común está representado por el jugador cuando pulsa un botón del *joystick*, pero también se pueden identificar eventos a nivel interno, como por ejemplo la reaparición o *respawn* de un NPC en el juego.

Gran parte de los motores de juegos incluyen un subsistema específico para el tratamiento de eventos, permitiendo al resto de componentes del motor o incluso a entidades específicas registrarse como partes interesadas en un determinado tipo de eventos. Este planteamiento está muy estrechamente relacionado con el patrón *Observer*.

Canales de eventos

Con el objetivo de independizar los publicadores y los suscriptores de eventos, se suele utilizar el concepto de *canal de eventos* como mecanismo de abstracción.

El tratamiento de eventos es un aspecto transversal a otras arquitecturas diseñadas para tratar el bucle de juego, por lo que es bastante común integrarlo dentro de otros esquemas más generales, como por ejemplo el que se discute a continuación y que está basado en la gestión de distintos estados dentro del juego.

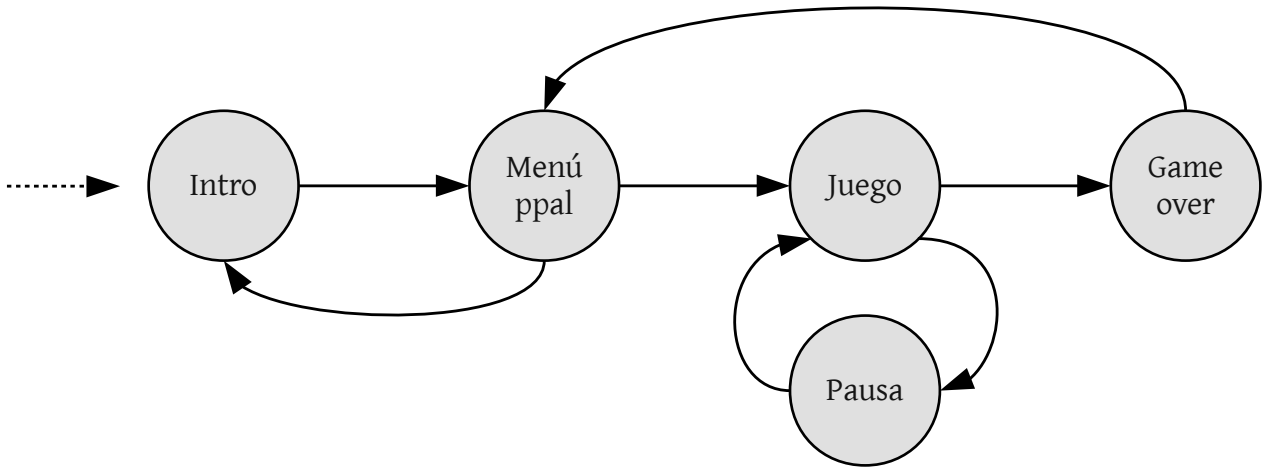


Figura 6.5: Visión general de una máquina de estados finita que representa los estados más comunes en cualquier juego.

Esquema basado en estados

Desde un punto de vista general, los juegos se pueden dividir en una serie de etapas o estados que se caracterizan no sólo por su funcionamiento sino también por la interacción con el usuario o jugador. Típicamente, en la mayor parte de los juegos es posible diferenciar los siguientes estados:

- **Introducción** o presentación, en la que se muestra al usuario aspectos generales del juego, como por ejemplo la temática del mismo o incluso cómo jugar.
- **Menú principal**, en la que el usuario ya puede elegir entre los distintos modos de juegos y que, normalmente, consiste en una serie de entradas textuales identificando las opciones posibles.
- **Juego**, donde ya es posible interactuar con la propia aplicación e ir completando los objetivos marcados.
- **Finalización** o *game over*, donde se puede mostrar información sobre la partida previamente jugada.

Evidentemente, esta clasificación es muy general ya que está planteada desde un punto de vista muy abstracto. Por ejemplo, si consideramos aspectos más específicos como por ejemplo el uso de dispositivos como *PlayStation Move*™, *Wiimote*™o *Kinect*™, sería necesario incluir un estado de calibración antes de poder utilizar estos dispositivos de manera satisfactoria.

Finite-state machines

Las máquinas de estados o autómatas representan modelos matemáticos utilizados para diseñar programas y lógica digital. En el caso del desarrollo de videojuegos se pueden usar para modelar diagramas de estados para, por ejemplo, definir los distintos comportamientos de un personaje.

Por otra parte, existe una relación entre cada uno de estos estados que se manifiesta en forma de **transiciones** entre los mismos. Por ejemplo, desde el estado de *introducción* sólo será posible acceder al estado de *menú principal*, pero no será posible acceder al resto de estados. En otras palabras, existirá una transición que va desde *introducción* a *menú principal*. Otro ejemplo podría ser la transición existente entre *finalización* y *menú principal*.

Este planteamiento basado en estados también debería poder manejar varios estados de manera simultánea para, por ejemplo, contemplar situaciones en las que sea necesario ofrecer algún tipo de menú sobre el propio juego en cuestión.

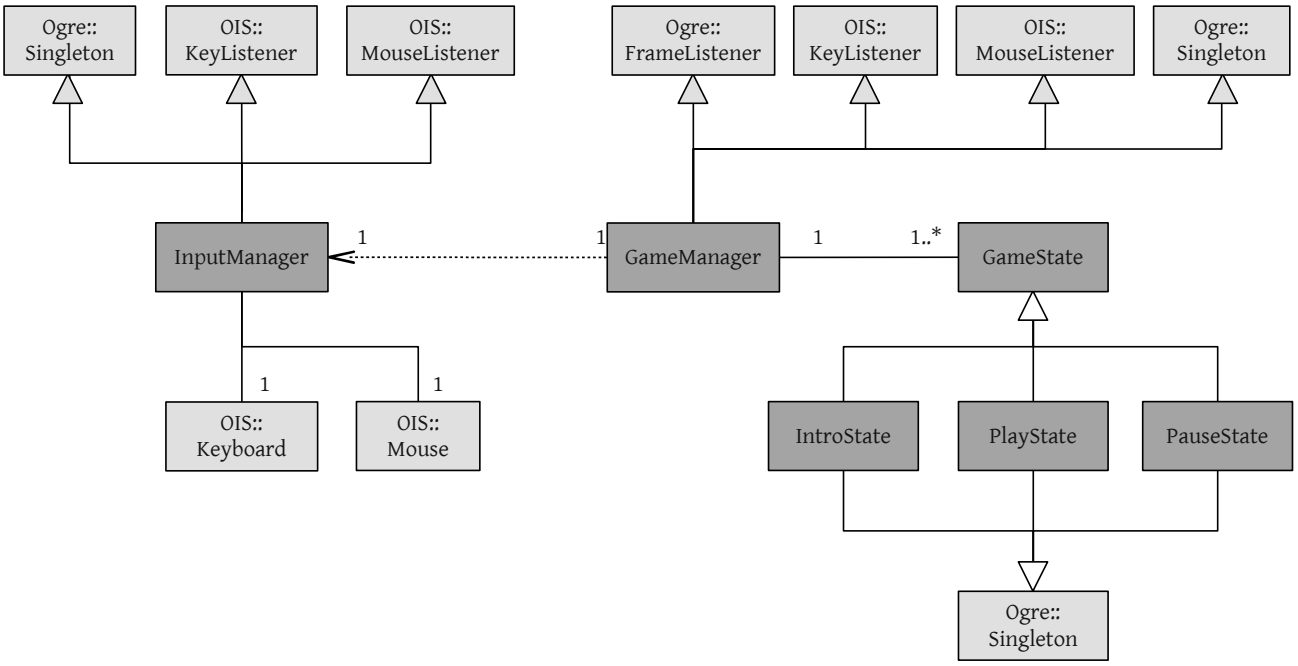


Figura 6.6: Diagrama de clases del esquema de gestión de estados de juego con Ogre3D. En un tono más oscuro se reflejan las clases específicas de dominio.

En la siguiente sección se discute en profundidad un caso de estudio en el que se utiliza Ogre3D para implementar un sencillo mecanismo basado en la gestión de estados. En dicha discusión se incluye un gestor responsable de capturar los eventos externos, como por ejemplo las pulsaciones de teclado o la interacción mediante el ratón.

6.1.4. Gestión de estados de juego con Ogre3D

Como ya se ha comentado, los juegos normalmente atraviesan una serie de estados durante su funcionamiento normal. En función del tipo de juego y de sus características, el número de estados variará significativamente. Sin embargo, es posible plantear un esquema común, compartido por todos los estados de un juego, que sirva para definir un modelo de gestión general, tanto para la interacción con los estados como para las transiciones existentes entre ellos.

La solución discutida en esta sección[1], que a su vez está basada en el artículo *Managing Game States in C++*. se basa en definir una clase abstracta, **GameState**, que contiene una serie de funciones virtuales a implementar en los estados específicos de un juego.

[1]La solución discutida aquí se basa en la planteada en el WiKi de Ogre3D, disponible en `http://www.ogre3d.org/tikiwiki/Managing+Game+States+with+OGRE`

El siguiente listado de código muestra el esqueleto de la clase *GameState* implementada en C++. Como se puede apreciar, esta clase es abstracta ya que mantiene una serie de funciones miembro como virtuales puras, con el objetivo de forzar su implementación en las clases que deriven de ella y que, por lo tanto, definan estados específicos de un juego. Estas funciones virtuales puras se pueden dividir en tres grandes bloques:

1. **Gestión básica del estado** (líneas 19-22), para definir qué hacer cuando se entra, sale, pausa o reanuda el estado.
2. **Gestión básica de tratamiento de eventos** (líneas 26-33), para definir qué hacer cuando se recibe un evento de teclado o de ratón.
3. **Gestión básica de eventos antes y después del renderizado** (líneas 37-38), operaciones típicas de la clase *Ogre::FrameListener*.

Clase abstracta en C++

En C++, las clases abstractas se definen mediante funciones virtuales puras y sirven para explicitar el contrato funcional entre una clase y sus clases derivadas.

Adicionalmente, existe otro bloque de funciones relativas a la **gestión básica de transiciones** (líneas 41-48), con operaciones para cambiar de estado, añadir un estado a la pila de estados y volver a un estado anterior, respectivamente. Las transiciones implican una interacción con la entidad *GameManager*, que se discutirá posteriormente.

La figura 6.6 muestra la relación de la clase *GameState* con el resto de clases, así como tres posibles especializaciones de la misma. Como se puede observar, esta clase está relacionada con *GameManager*, responsable de la gestión de los distintos estados y de sus transiciones.

Listado 6.4: Clase GameState.

```
1  #ifndef GameState_H
2  #define GameState_H
3
4  #include <Ogre.h>
5  #include <OIS/OIS.h>
6
7  #include "GameManager.h"
8  #include "InputManager.h"
9
10 // Clase abstracta de estado básico.
11 // Definición base sobre la que extender
12 // los estados del juego.
13 class GameState {
14
15  public:
16   GameState() {}
17
18   // Gestión básica del estado.
19   virtual void enter () = 0;
20   virtual void exit () = 0;
21   virtual void pause () = 0;
22   virtual void resume () = 0;
23
24   // Gestión básica para el tratamiento
25   // de eventos de teclado y ratón.
26   virtual void keyPressed (const OIS::KeyEvent &e) = 0;
27   virtual void keyReleased (const OIS::KeyEvent &e) = 0;
28
29   virtual void mouseMoved (const OIS::MouseEvent &e) = 0;
30   virtual void mousePressed (const OIS::MouseEvent &e,
31                    OIS::MouseButtonID id) = 0;
32   virtual void mouseReleased (const OIS::MouseEvent &e,
33                     OIS::MouseButtonID id) = 0;
34
35   // Gestión básica para la gestión
```

```
36   // de eventos antes y después de renderizar un frame.
37   virtual bool frameStarted (const Ogre::FrameEvent& evt) = 0;
38   virtual bool frameEnded (const Ogre::FrameEvent& evt) = 0;
39
40   // Gestión básica de transiciones.
41   void changeState (GameState* state) {
42     GameManager::getSingletonPtr()->changeState(state);
43   }
44   void pushState (GameState* state) {
45     GameManager::getSingletonPtr()->pushState(state);
46   }
47   void popState () {
48     GameManager::getSingletonPtr()->popState();
49   }
50
51 };
52
53 #endif
```

Sin embargo, antes de pasar a discutir esta clase, en el diseño discutido se contempla la definición explícita de la clase **InputManager**, como punto central para la gestión de eventos de entrada, como por ejemplo los de teclado o de ratón.

¿Ogre::Singleton?

La clase *InputManager* implementa el patrón *Singleton* mediante las utilidades de Ogre3D. Es posible utilizar otros esquemas para que su implementación no depende de Ogre y se pueda utilizar con otros *frameworks*.

El *InputManager* sirve como interfaz para aquellas entidades que estén interesadas en procesar eventos de entrada (como se discutirá más adelante), ya que mantiene operaciones para añadir y eliminar *listeners* de dos tipos: i) *OIS::KeyListener* y ii) *OIS::MouseListener*. De hecho, esta clase hereda de ambas clases. Además, implementa el patrón *Singleton* con el objetivo de que sólo exista una única instancia de la misma.

Listado 6.5: Clase InputManager.

```
1  // SE OMITE PARTE DEL CÓDIGO FUENTE.
2  // Gestor para los eventos de entrada (teclado y ratón).
3  class InputManager : public Ogre::Singleton<InputManager>,
4    public OIS::KeyListener, public OIS::MouseListener {
5   public:
6    InputManager ();
7    virtual ~InputManager ();
8
9    void initialise (Ogre::RenderWindow *renderWindow);
10   void capture ();
11
12   // Gestión de listeners.
13   void addKeyListener (OIS::KeyListener *keyListener,
14               const std::string& instanceName);
15   void addMouseListener (OIS::MouseListener *mouseListener,
16                 const std::string& instanceName );
17   void removeKeyListener (const std::string& instanceName);
18   void removeMouseListener (const std::string& instanceName);
19   void removeKeyListener (OIS::KeyListener *keyListener);
20   void removeMouseListener (OIS::MouseListener *mouseListener);
21
22   OIS::Keyboard* getKeyboard ();
23   OIS::Mouse* getMouse ();
24
25   // Heredados de Ogre::Singleton.
26   static InputManager& getSingleton ();
27   static InputManager* getSingletonPtr ();
28
29  private:
30   // Tratamiento de eventos.
31   // Delegará en los listeners.
32   bool keyPressed (const OIS::KeyEvent &e);
33   bool keyReleased (const OIS::KeyEvent &e);
34
35   bool mouseMoved (const OIS::MouseEvent &e);
```

```
36   bool mousePressed (const OIS::MouseEvent &e,
37              OIS::MouseButtonID id);
38   bool mouseReleased (const OIS::MouseEvent &e,
39               OIS::MouseButtonID id);
40
41   OIS::InputManager *_inputSystem;
42   OIS::Keyboard *_keyboard;
43   OIS::Mouse *_mouse;
44   std::map<std::string, OIS::KeyListener*> _keyListeners;
45   std::map<std::string, OIS::MouseListener*> _mouseListeners;
46 };
```

En la sección privada de *InputManager* se declaran funciones miembro (líneas 40-47) que se utilizarán para notificar a los *listeners* registrados en el *InputManager* acerca de la existencia de eventos de entrada, tanto de teclado como de ratón. Este planteamiento garantiza la escalabilidad respecto a las entidades interesadas en procesar eventos de entrada. Las estructuras de datos utilizadas para almacenar los distintos tipos de *listeners* son elementos de la clase *map* de la biblioteca estándar de C++, indexando los mismos por el identificador textual asociado a los *listeners*.

Por otra parte, es importante destacar las variables miembro *_keyboard* y *_mouse* que se utilizarán para capturar los eventos de teclado y ratón, respectivamente. Dicha captura se realizará en cada *frame* mediante la función *capture()*, definida tanto en el *InputManager* como en *OIS::Keyboard* y *OIS::Mouse*.

El siguiente listado de código muestra la clase **GameManager**, que representa a la entidad principal de gestión del esquema basado en estados que se discute en esta sección. Esta clase es una derivación de de *Ogre::Singleton* para manejar una única instancia y de tres clases clave para el tratamiento de eventos. En concreto, *GameManager* hereda de los dos tipos de *listeners* previamente comentados con el objetivo de permitir el registro con el *InputManager*.

Listado 6.6: Clase GameManager.

```
1  // SE OMITE PARTE DEL CÓDIGO FUENTE.
2  class GameState;
3
4  class GameManager : public Ogre::FrameListener, public Ogre::
       Singleton<GameManager>,
5    public OIS::KeyListener, public OIS::MouseListener
6  {
7   public:
8    GameManager ();
9    ~GameManager (); // Limpieza de todos los estados.
10
11   // Para el estado inicial.
12   void start (GameState* state);
13
14   // Funcionalidad para transiciones de estados.
15   void changeState (GameState* state);
16   void pushState (GameState* state);
17   void popState ();
18
19   // Heredados de Ogre::Singleton...
20
21  protected:
22   Ogre::Root* _root;
23   Ogre::SceneManager* _sceneManager;
24   Ogre::RenderWindow* _renderWindow;
25
26   // Funciones de configuración.
27   void loadResources ();
28   bool configure ();
29
30   // Heredados de FrameListener.
```

```
31   bool frameStarted (const Ogre::FrameEvent& evt);
32   bool frameEnded (const Ogre::FrameEvent& evt);
33
34  private:
35   // Funciones para delegar eventos de teclado
36   // y ratón en el estado actual.
37   bool keyPressed (const OIS::KeyEvent &e);
38   bool keyReleased (const OIS::KeyEvent &e);
39
40   bool mouseMoved (const OIS::MouseEvent &e);
41   bool mousePressed (const OIS::MouseEvent &e, OIS::MouseButtonID
         id);
42   bool mouseReleased (const OIS::MouseEvent &e, OIS::MouseButtonID
         id);
43
44   // Gestor de eventos de entrada.
45   InputManager *_inputMgr;
46   // Estados del juego.
47   std::stack<GameState*> _states;
48 };
```

Note que esta clase contiene una función miembro *start()* (línea 12), definida de manera explícita para inicializar el gestor de juego, establecer el estado inicial (pasado como parámetro) y arrancar el bucle de renderizado.

Listado 6.7: Clase GameManager. Función start().

```
1  void
2  GameManager::start
3  (GameState* state)
4  {
5    // Creación del objeto Ogre::Root.
6    _root = new Ogre::Root();
7
8    if (!configure())
9      return;
10
11   loadResources();
12
13   _inputMgr = new InputManager;
14   _inputMgr->initialise(_renderWindow);
15
16   // Registro como key y mouse listener...
17   _inputMgr->addKeyListener(this, "GameManager");
18   _inputMgr->addMouseListener(this, "GameManager");
19
20   // El GameManager es un FrameListener.
21   _root->addFrameListener(this);
22
23   // Transición al estado inicial.
24   changeState(state);
25
26   // Bucle de rendering.
27   _root->startRendering();
28 }
```

La clase *GameManager* mantiene una estructura de datos denominada *_states* que refleja las transiciones entre los diversos estados del juego. Dicha estructura se ha implementado mediante una pila (clase *stack* de STL) ya que refleja fielmente la naturaleza de cambio y gestión de estados. Para cambiar de un estado A a otro B, suponiendo que A sea la cima de la pila, habrá que realizar las operaciones siguientes:

1. Ejecutar *exit()* sobre A.
2. Desapilar A.

3. Apilar B (pasaría a ser el estado activo).
4. Ejecutar *enter()* sobre B.

El siguiente listado de código muestra una posible implementación de la función *changeState()* de la clase *GameManager*. Note cómo la estructura de pila de estados permite un acceso directo al estado actual (cima) para llevar a cabo las operaciones de gestión necesarias. Las transiciones se realizan con las típicas operaciones de *push* y *pop*.

Listado 6.8: Clase GameManager. Función changeState().

```
1  void
2  GameManager::changeState
3  (GameState* state)
4  {
5    // Limpieza del estado actual.
6    if (!_states.empty()) {
7      // exit() sobre el último estado.
8      _states.top()->exit();
9      // Elimina el último estado.
10     _states.pop();
11   }
12   // Transición al nuevo estado.
13   _states.push(state);
14   // enter() sobre el nuevo estado.
15   _states.top()->enter();
16 }
```

Otro aspecto relevante del diseño de esta clase es la delegación de eventos de entrada asociados a la interacción por parte del usuario con el teclado y el ratón. El diseño discutido permite delegar directamente el tratamiento del evento al estado activo, es decir, al estado que ocupe la cima de la pila. Del mismo modo, se traslada a dicho estado la implementación de las funciones *frameStarted()* y *frameEnded()*. El siguiente listado de código muestra cómo la implementación de, por ejemplo, la función *keyPressed()* es trivial.

Listado 6.9: Clase GameManager. Función keyPressed().

```
1  bool
2  GameManager::keyPressed
3  (const OIS::KeyEvent &e)
4  {
5    _states.top()->keyPressed(e);
6    return true;
7  }
```

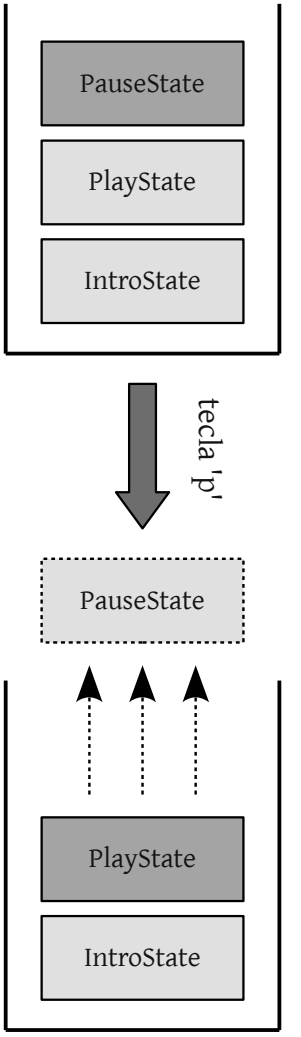


Figura 6.7: Actualización de la pila de estados para reanudar el juego (evento teclado 'p').

6.1.5. Definición de estados concretos

Este esquema de gestión de estados general, el cual contiene una clase genérica *GameState*, permite la definición de estados específicos vinculados a un juego en particular. En la figura 6.6 se muestra gráficamente cómo la clase *GameState* se extiende para definir tres estados:

- **IntroState**, que define un estado de introducción o inicialización.
- **PlayState**, que define el estado principal de un juego y en el que se desarrollará la lógica del mismo.

Figura 6.8: Capturas de pantalla del juego *Supertux*. Arriba, estado de introducción; medio, estado de juego; abajo, estado de pausa.

Figura 6.9: Pantalla de presentación del juego *Frozen Bubble*.

- **PauseState**, que define un estado de pausa típico en cualquier tipo de juego.

La implementación de estos estados permite hacer explícito el comportamiento del juego en cada uno de ellos, al mismo tiempo que posibilita las transiciones entre los mismos. Por ejemplo, una transición típica será pasar del estado *PlayState* al estado *PauseState*.

A la hora de llevar a cabo dicha implementación, se ha optado por utilizar el patrón *Singleton*, mediante la clase *Ogre::Singleton*, para garantizar que sólo existe una instacia de un objeto por estado.

El siguiente listado de código muestra una posible declaración de la clase *IntroState*. Como se puede apreciar, esta clase hace uso de las funciones típicas para el tratamiento de eventos de teclado y ratón.

Listado 6.10: Clase IntroState.

```
1  // SE OMITE PARTE DEL CÓDIGO FUENTE.
2  class IntroState : public Ogre::Singleton<IntroState>,
3                     public GameState
4  {
5   public:
6    IntroState() {}
7
8    void enter (); void exit ();
9    void pause (); void resume ();
10
11   void keyPressed (const OIS::KeyEvent &e);
12   void keyReleased (const OIS::KeyEvent &e);
13   // Tratamiento de eventos de ratón...
14   // frameStarted(), frameEnded()..
15
16   // Heredados de Ogre::Singleton.
17   static IntroState& getSingleton ();
18   static IntroState* getSingletonPtr ();
19
20  protected:
21   Ogre::Root* _root;
22   Ogre::SceneManager* _sceneMgr;
23   Ogre::Viewport* _viewport;
24   Ogre::Camera* _camera;
25   bool _exitGame;
26 };
```

Así mismo, también especifíca la funcionalidad vinculada a la gestión básica de estados. Recuerde que, al extender la clase *GameState*, los estados específicos han de cumplir el contrato funcional definido por dicha clase abstracta.

Finalmente, la **transición de estados** se puede gestionar de diversas formas. Por ejemplo, es perfectamente posible realizarla mediante los eventos de teclado. En otras palabras, la pulsación de una tecla en un determinado estado sirve como disparador para pasar a otro estado. Recuerde que el paso de un estado a otro puede implicar la ejecución de las funciones *exit()* o *pause()*, dependiendo de la transición concreta.

El siguiente listado de código muestra las transiciones entre el estado de juego y el estado de pausa y entre éste último y el primero, es decir, la acción comúnmente conocida como *reanudación* (*resume*). Para ello, se hace uso de la tecla 'p'.

Listado 6.11: Transiciones entre estados.

```
1 void PlayState::keyPressed (const OIS::KeyEvent &e) {
2   if (e.key == OIS::KC_P) // Tecla p ->PauseState.
3     pushState(PauseState::getSingletonPtr());
```

```
4 }
5
6 void PauseState::keyPressed (const OIS::KeyEvent &e) {
7   if (e.key == OIS::KC_P) // Tecla p ->Estado anterior.
8     popState();
9 }
```

Note cómo la activación del estado de pausa provoca apilar dicho estado en la pila (*pushState()* en la línea 7), mientras que la reanudación implica desapilarlo (*popState()* en la línea 16).

6.2. Gestión básica de recursos

En esta sección se discute la gestión básica de los recursos, haciendo especial hincapié en dos casos de estudio concretos: i) la gestión básica del **sonido** y ii) el **sistema de archivos**. Antes de llevar a cabo un estudio específico de estas dos cuestiones, la primera sección de este capítulo introduce la problemática de la gestión de recursos en los motores de juegos. Así mismo, se discuten las posibilidades que el *framework* Ogre3D ofrece para gestionar dicha problemática.

Figura 6.10: Captura de pantalla del videojuego *Lemmings* 2. Nunca la gestión de recursos fue tan importante...

En el caso de la gestión básica del sonido, el lector será capaz de manejar una serie de abstracciones, en torno a la biblioteca multimedia SDL, para integrar música y efectos de sonido en los juegos que desarrolle. Por otra parte, en la sección relativa a la gestión del sistema de archivos se planteará la problemática del tratamiento de archivos y se estudiarán técnicas de entrada/salida asíncrona.

Debido a la naturaleza multimedia de los motores de juegos, una consecuencia directa es la necesidad de gestionar distintos tipos de datos, como por ejemplo geometría tridimensional, texturas, animaciones, sonidos, datos relativos a la gestión de física y colisiones, etc. Evidentemente, esta naturaleza tan variada se ha de gestionar de forma **consistente** y garantizando la integridad de los datos.

Por otra parte, las potenciales limitaciones hardware de la plataforma sobre la que se ejecutará el juego implica que sea necesario plantear un mecanismo eficiente para cargar y liberar los recursos asociados a dichos datos multimedia. Por lo tanto, una máxima del motor de juegos es asegurarse que solamente existe una copia en memoria de un determinado recurso multimedia, de manera independiente al número de instancias que lo estén utilizando en un determinado momento. Este esquema permite optimizar los recursos y manejarlos de una forma adecuada.

Media manager

El gestor de recursos o *resource manager* se suele denominar comúnmente *media manager* o *asset manager*.

Un ejemplo representativo de esta cuestión está representado por las mallas poligonales y las texturas. Si, por ejemplo, siete mallas comparten la misma textura, es decir, la misma imagen bidimensional, entonces es deseable mantener una única copia de la textura en memoria principal, en lugar de mantener siete. En este contexto, la mayoría de motores de juegos integran algún tipo de **gestor de recursos** para cargar y gestionar la diversidad de recursos que se utilizan en los juegos de última generación.

6.2.1. Gestión de recursos con Ogre3D

Ogre facilita la carga, liberación y gestión de recursos mediante un planteamiento bastante sencillo centrado en optimizar el consumo de memoria de la aplicación gráfica. Recuerde que la memoria es un recurso escaso en el ámbito del desarrollo de videojuegos.

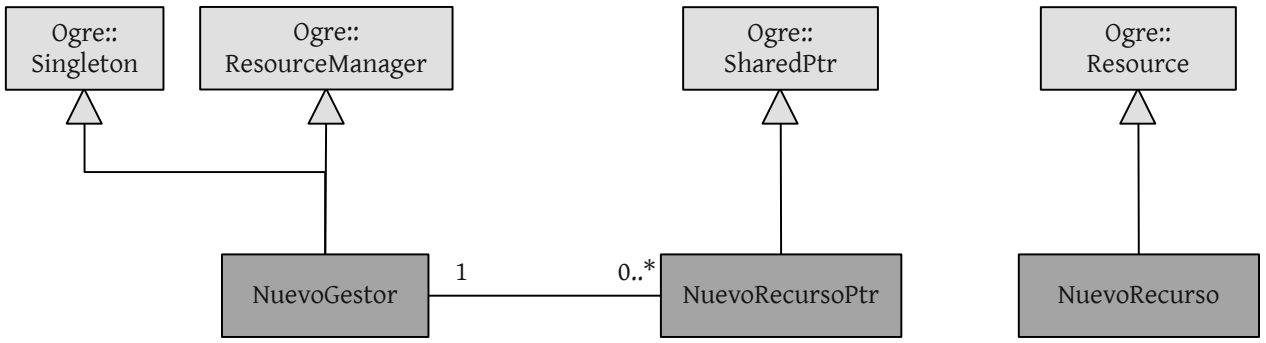


Figura 6.11: Diagrama de clases de las principales clases utilizadas en Ogre3D para la gestión de recursos. En un tono más oscuro se reflejan las clases específicas de dominio.

Una de las ideas fundamentales de la carga de recursos se basa en almacenar en memoria principal una única copia de cada recurso. Éste se puede utilizar para representar o gestionar múltiples entidades.

Carga de niveles

Una de las técnicas bastantes comunes a la hora de cargar niveles de juego consiste en realizarla de manera previa al acceso a dicho nivel. Por ejemplo, se pueden utilizar estructuras de datos para representar los puntos de acceso de un nivel al siguiente para adelantar el proceso de carga. El mismo planteamiento se puede utilizar para la carga de texturas en escenarios.

Debido a las restricciones hardware que una determinada plataforma de juegos puede imponer, resulta fundamental hacer uso de un mecanismo de gestión de recursos que sea flexible y escalable para garantizar el diseño y gestión de cualquier tipo de recurso, junto con la posibilidad de cargarlo y liberarlo en tiempo de ejecución, respectivamente. Por ejemplo, si piensa en un juego de plataformas estructurado en diversos niveles, entonces no sería lógico hacer una carga inicial de todos los niveles al iniciar el juego. Por el contrario, sería más adecuado cargar un nivel cuando el jugador vaya a acceder al mismo.

En Ogre, la gestión de recursos se lleva a cabo utilizando principalmente cuatro clases (ver figura 6.11):

- **Ogre::Singleton**, con el objetivo de garantizar que sólo existe una instancia de un determinado recurso o gestor.
- **Ogre::ResourceManager**, con el objetivo de centralizar la gestión del *pool* de recursos de un determinado tipo.
- **Ogre::Resource**, entidad que representa a una clase abstracta que se puede asociar a recursos específicos.
- **Ogre::SharedPtr**, clase que permite la gestión *inteligente* de recursos que necesitan una destrucción implícita.

Básicamente, el desarrollador que haga uso del planteamiento que Ogre ofrece para la gestión de recursos tendrá que implementar algunas funciones heredadas de las clases anteriormente mencionadas, completando así la implementación específica del dominio, es decir, específica de los recursos a gestionar. Más adelante se discute en detalle un ejemplo relativo a los recursos de sonido.

6.2.2. Gestión básica del sonido

Introducción a SDL

SDL (Simple Directmedia Layer) es una **biblioteca multimedia** y multiplataforma ampliamente utilizada en el ámbito del desarrollo de aplicaciones multimedia. Desde un punto de vista funcional, SDL proporciona una serie de APIs para el manejo de vídeo, audio, eventos de entrada, multi-hilo o renderizado con OpenGL, entre otros aspectos. Desde un punto de vista abstracto, SDL proporciona una API consistente de manera independiente a la plataforma de desarrollo.

Figura 6.12: Logo de la biblioteca multiplataforma *Simple Directmedia Layer*.

SDL está bien estructurado y es fácil de utilizar. La filosofía de su diseño se puede resumir en ofrecer al desarrollador **diversas herramientas** que se pueden utilizar de manera independiente, en lugar de manejar una biblioteca software de mayor envergadura. Por ejemplo, un juego podría hacer uso de la biblioteca SDL únicamente para la gestión de sonido, mientras que la parte gráfica sería gestionada de manera independiente (utilizando Ogre3D, por ejemplo).

SDL y *Civilization*

El *port* a sistemas GNU Linux del juego *Civilization* se realizó haciendo uso de SDL. En este caso particular, los personajes se renderizaban haciendo uso de superficies.

SDL se puede integrar perfectamente con **OpenGL** para llevar a cabo la inicialización de la parte gráfica de una aplicación interactiva, delegando en SDL el propio tratamiento de los eventos de entrada. Hasta ahora, en el módulo 2 (*Programación Gráfica*) se había utilizado la biblioteca GLUT (OpenGL Utility Toolkit) para implementar programas que hicieran uso de OpenGL. Sin embargo, SDL proporciona un gran número de **ventajas** con respecto a esta alternativa[2]:

- SDL fue diseñado para programadores de videojuegos.
- Es modular, simple y portable.
- Proporciona soporte para la gestión de eventos, la gestión del tiempo, el manejo de sonido, la integración con dispositivos de almacenamiento externos como por ejemplo CDs, el renderizado con OpenGL e incluso la gestión de red (*networking*).

Event handlers

SDL posibilita la definición de arquitecturas multi-capa para el tratamiento de eventos, facilitando así su delegación por parte del código dependiente del dominio.

El concepto de **superficie** como estructura de datos es esencial en SDL, ya que posibilita el tratamiento de la parte gráfica. En esencia, una superficie representa un bloque de memoria utilizado para almacenar una región rectangular de píxeles. El principal objetivo de diseño es agilizar la copia de superficies tanto como sea posible. Por ejemplo, una superficie se puede utilizar para renderizar los propios caracteres de un juego.

El siguiente listado de código muestra un ejemplo de integración de OpenGL en SDL, de manera que SDL oculta todos los aspectos dependientes de la plataforma a la hora de inicializar el núcleo de OpenGL. Es importante resaltar que OpenGL toma el control del subsistema de vídeo de SDL. Sin embargo, el resto de componentes de SDL no se ven afectados por esta cuestión.

La instalación de SDL en sistemas Debian y derivados es trivial mediante los siguientes comandos:

```
$ sudo apt-get update
$ sudo apt-get install libsdl1.2-dev
$ sudo apt-get install libsdl-image1.2-dev
$ sudo apt-get install libsdl-sound1.2-dev libsdl-mixer1.2-dev
```

Figura 6.13: Una vez más, el sistema de paquetes de Debian GNU/Linux y distros derivadas simplifica enormemente la instalación y configuración de bibliotecas de desarrollo. *Long live Debian!*

Para generar un fichero ejecutable, simplemente es necesario enlazar con las bibliotecas necesarias.

[2] GLUT fue concebido para utilizarse en un entorno más académico, simplificando el tratamiento de eventos y la gestión de ventanas.

Listado 6.12: Ejemplo sencillo SDL + OpenGL.

```
1  #include <SDL/SDL.h>
2  #include <GL/gl.h>
3  #include <stdio.h>
4
5  int main (int argc, char *argv[]) {
6    SDL_Surface *screen;
7    // Inicialización de SDL.
8    if (SDL_Init(SDL_INIT_VIDEO) != 0) {
9      fprintf(stderr, "Unable to initialize SDL: %s\n",
10         SDL_GetError());
11     return -1;
12   }
13
14   // Cuando termine el programa, llamada a SQLQuit().
15   atexit(SDL_Quit);
16   // Activación del double buffering.
17   SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);
18
19   // Establecimiento del modo de vídeo con soporte para OpenGL.
20   screen = SDL_SetVideoMode(640, 480, 16, SDL_OPENGL);
21   if (screen == NULL) {
22     fprintf(stderr, "Unable to set video mode: %s\n",
23         SDL_GetError());
24     return -1;
25   }
26
27   SDL_WM_SetCaption("OpenGL with SDL!", "OpenGL");
28   // ¡Ya es posible utilizar comandos OpenGL!
29   glViewport(80, 0, 480, 480);
30
31   glMatrixMode(GL_PROJECTION);
32   glLoadIdentity();
33   glFrustum(-1.0, 1.0, -1.0, 1.0, 1.0, 100.0);
34   glClearColor(1, 1, 1, 0);
35
36   glMatrixMode(GL_MODELVIEW); glLoadIdentity();
37   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
38
39   // Renderizado de un triángulo.
40   glBegin(GL_TRIANGLES);
41   glColor3f(1.0, 0.0, 0.0); glVertex3f(0.0, 1.0, -2.0);
42   glColor3f(0.0, 1.0, 0.0); glVertex3f(1.0, -1.0, -2.0);
43   glColor3f(0.0, 0.0, 1.0); glVertex3f(-1.0, -1.0, -2.0);
44   glEnd();
45
46   glFlush();
47   SDL_GL_SwapBuffers(); // Intercambio de buffers.
48   SDL_Delay(5000);      // Espera de 5 seg.
49
50   return 0;
51 }
```

Como se puede apreciar en el listado anterior, la línea 20 establece el modo de vídeo con soporte para OpenGL para, posteriormente, hacer uso de comandos OpenGL a partir de la línea 29. Sin embargo, note cómo el intercambio del contenido entre el *front buffer* y el *back buffer* se realiza en la línea 47 mediante una primitiva de SDL.

Listado 6.13: Ejemplo de Makefile para integrar SDL.

```
1 CFLAGS := -c -Wall
2 LDFLAGS := `sdl-config --cflags --libs` -lSDL_image -lGL
3 LDLIBS := -lSDL_image -lGL
4 CC := gcc
5
6 all: basic_sdl_opengl
7
```

```
 8 basic_sdl_opengl: basic_sdl_opengl.o
 9     $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)
10
11 basic_sdl_opengl.o: basic_sdl_opengl.c
12     $(CC) $(CFLAGS) $^ -o $@
13
14 clean:
15     @echo Cleaning up...
16     rm -f *~ *.o basic_sdl_opengl
17     @echo Done.
18
19 vclean: clean
```

Reproducción de música

En esta sección se discute cómo implementar[3] un nuevo recurso que permita la reproducción de archivos de música dentro de un juego. En este ejemplo se hará uso de la biblioteca *SDL_mixer* para llevar a cabo la reproducción de archivos de sonido[4].

En primer lugar se define una clase **Track** que será utilizada para gestionar el recurso asociado a una canción. Como ya se ha comentado anteriormente, esta clase hereda de la clase *Ogre::Resource*, por lo que será necesario implementar las funciones necesarias para el nuevo tipo de recurso. Además, se incluirá la funcionalidad típica asociada a una canción, como las clásicas operaciones *play*, *pause* o *stop*. El siguiente listado de código muestra la declaración de la clase *Track*.

Figura 6.14: La manipulación básica de sonido incluye las típicas operaciones de *play*, *pause* y *stop*.

Listado 6.14: Clase Track.

```
 1 #include <SDL/SDL_mixer.h>
 2 #include <OGRE/Ogre.h>
 3
 4 class Track : public Ogre::Resource {
 5  public:
 6   // Constructor (ver Ogre::Resource).
 7   Track (Ogre::ResourceManager* pManager,
 8      const Ogre::String& resource_name,
 9      Ogre::ResourceHandle handle,
10      const Ogre::String& resource_group,
11      bool manual_load = false,
12      Ogre::ManualResourceLoader* pLoader = 0);
13   ~Track ();
14
15   // Manejo básico del track.
16   void play (int loop = -1);
17   void pause ();
18   void stop ();
19
20   void fadeIn (int ms, int loop);
21   void fadeOut (int ms);
22   static bool isPlaying ();
23
24  private:
25   // Funcionalidad de Ogre::Resource.
26   void loadImpl ();
27   void unloadImpl ();
28   size_t calculateSize () const;
29
30   // Variables miembro.
```

[3] La implementación de los recursos de sonido está basada en el artículo titulado *Extender la gestión de recursos, audio* del portal *IberOgre*, el cual se encuentra disponible en `http://osl2.uca.es/iberogre/index.php/Extender_la_gestión_de_recursos,_audio`.

[4] *SDL_mixer* sólo permite la reproducción de un *clip* de sonido de manera simultánea, por lo que no será posible realizar mezclas de canciones.

```
31   Mix_Music* _pTrack; // SDL
32   Ogre::String _path; // Ruta al track.
33   size_t _size;       // Tamaño.
34 };
```

El constructor de esta clase viene determinado por la especificación del constructor de la clase *Ogre::Resource*. De hecho, su implementación delega directamente en el constructor de la clase padre para poder instanciar un recurso, además de inicializar las propias variables miembro.

Por otra parte, las funciones de manejo básico del *track* (líneas 16-22) son en realidad una interfaz para manejar de una manera adecuada la biblioteca *SDL_mixer*.

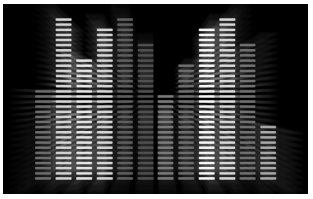


Figura 6.15: Es posible incluir efectos de sonidos más avanzados, como por ejemplo reducir el nivel de volumen a la hora de finalizar la reproducción de una canción.

Por ejemplo, la función miembro *play* simplemente interactúa con SDL para comprobar si la canción estaba pausada y, en ese caso, reanudarla. Si la canción no estaba pausada, entonces dicha función la reproduce desde el principio. Note cómo se hace uso del gestor de *logs* de Ogre3D para almacenar en un archivo la posible ocurrencia de algún tipo de error.

Listado 6.15: Clase Track. Función play().

```
1  void
2  Track::play
3  (int loop)
4  {
5    Ogre::LogManager* pLogManager =
6      Ogre::LogManager::getSingletonPtr();
7
8    if(Mix_PausedMusic()) // Estaba pausada?
9      Mix_ResumeMusic();  // Reanudación.
10
11   // Si no, se reproduce desde el principio.
12   else {
13     if (Mix_PlayMusic(_pTrack, loop) == -1) {
14       pLogManager->logMessage("Track::play() Error al....");
15       throw (Ogre::Exception(Ogre::Exception::ERR_FILE_NOT_FOUND,
16                  "Imposible reproducir...",
17                  "Track::play()"));
18         }
19     }
20 }
```

Dos de las funciones más importantes de cualquier tipo de recurso que extienda de *Ogre::Resource* son *loadImpl()* y *unloadImpl()*, utilizadas para cargar y liberar el recurso, respectivamente. Obviamente, cada tipo de recurso delegará en la funcionalidad necesaria para llevar a cabo dichas tareas. En el caso de la gestión básica del sonido, estas funciones delegarán principalmente en *SDL_mixer*. A continuación se muestra el código fuente de dichas funciones.

Debido a la necesidad de manejar de manera eficiente los recursos de sonido, la solución discutida en esta sección contempla la definición de punteros *inteligentes* a través de la clase **TrackPtr**. La justificación de esta propuesta, como ya se introdujo anteriormente, consiste en evitar la duplicidad de recursos, es decir, en evitar que un mismo recurso esté cargado en memoria principal más de una vez.

Listado 6.16: Clase Track. Funciones loadImpl() y unloadImpl().

```
1  void
2  Track::loadImpl () // Carga del recurso.
3  {
4    // Ruta al archivo.
5    Ogre::FileInfoListPtr info;
6    info = Ogre::ResourceGroupManager::getSingleton().
```

```
 7     findResourceFileInfo(mGroup, mName);
 8
 9   for (Ogre::FileInfoList::const_iterator i = info->begin();
10        i != info->end(); ++i) {
11     _path = i->archive->getName() + "/" + i->filename;
12   }
13
14   if (_path == "") {   // Archivo no encontrado...
15     // Volcar en el log y lanzar excepción.
16   }
17
18   // Cargar el recurso de sonido.
19   if ((_pTrack = Mix_LoadMUS(_path.c_str())) == NULL) {
20     // Si se produce un error al cargar el recurso,
21     // volcar en el log y lanzar excepción.
22   }
23
24   // Cálculo del tamaño del recurso de sonido.
25   _size = ...
26 }
27
28 void
29 Track::unloadImpl()
30 {
31   if (_pTrack) {
32     // Liberar el recurso de sonido.
33     Mix_FreeMusic(_pTrack);
34   }
35 }
```

Ogre3D permite el uso de **punteros inteligentes** compartidos, definidos en la clase *Ogre::SharedPtr*, con el objetivo de parametrizar el recurso definido por el desarrollador, por ejemplo *Track*, y almacenar, internamente, un contador de referencias a dicho recurso. Básicamente, cuando el recurso se copia, este contador se incrementa. Si se destruye alguna referencia al recurso, entonces el contador se decrementa. Este esquema permite liberar recursos cuando no se estén utilizando en un determinado momento y compartir un mismo recurso entre varias entidades.

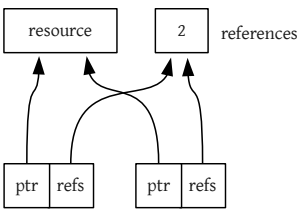


Figura 6.16: El uso de *smart pointers* optimiza la gestión de recursos y facilita su liberación.

El listado de código 6.17 muestra la implementación de la clase *TrackPtr*, la cual incluye una serie de funciones (básicamente constructores y asignador de copia) heredadas de *Ogre::SharedPtr*. A modo de ejemplo, también se incluye el código asociado al constructor de copia. Como se puede apreciar, en él se incrementa el contador de referencias al recurso.

Una vez implementada la lógica necesaria para instanciar y manejar recursos de sonido, el siguiente paso consiste en definir un gestor o *manager* específico para centralizar la administración del nuevo tipo de recurso. Ogre3D facilita enormemente esta tarea gracias a la clase *Ogre::ResourceManager*. En el caso particular de los recursos de sonido se define la clase **TrackManager**, cuyo esqueleto se muestra en el listado de código 6.18.

Esta clase no sólo hereda del gestor de recursos de Ogre, sino que también lo hace de la clase *Ogre::Singleton* con el objetivo de manejar una única instancia del gestor de recursos de sonido. Las funciones más relevantes son las siguientes:

- **load()** (líneas `10-11`), que permite la carga de canciones por parte del desarrollador. Si el recurso a cargar no existe, entonces lo creará internamente utilizando la función que se comenta a continuación.
- **createImpl()** (líneas `18-23`), función que posibilita la creación de un nuevo recurso, es decir, una nueva instancia de la clase *Track*. El desarrollador es responsable de realizar la carga del recurso una vez que ha sido creado.

Listado 6.17: Clase TrackPtr y constructor de copia.

```
// Smart pointer a Track.
class TrackPtr: public Ogre::SharedPtr<Track> {
 public:
  // Es necesario implementar constructores
  // y operador de asignación.
 TrackPtr(): Ogre::SharedPtr<Track>() {}
 explicit TrackPtr(Track* m): Ogre::SharedPtr<Track>(m) {}
 TrackPtr(const TrackPtr &m): Ogre::SharedPtr<Track>(m) {}
 TrackPtr(const Ogre::ResourcePtr &r);
 TrackPtr& operator= (const Ogre::ResourcePtr& r);
};

TrackPtr::TrackPtr
(const Ogre::ResourcePtr &resource):
  Ogre::SharedPtr<Track>()
{
  // Comprobar la validez del recurso.
  if (resource.isNull())
    return;

  // Para garantizar la exclusión mutua...
  OGRE_LOCK_MUTEX(*resource.OGRE_AUTO_MUTEX_NAME)
  OGRE_COPY_AUTO_SHARED_MUTEX(resource.OGRE_AUTO_MUTEX_NAME)

  pRep = static_cast<Track*>(resource.getPointer());
  pUseCount = resource.useCountPointer();
  useFreeMethod = resource.freeMethod();

  // Incremento del contador de referencias.
  if (pUseCount)
    ++(*pUseCount);
}
```

Con las tres clases que se han discutido en esta sección ya es posible realizar la carga de recursos de sonido, delegando en la biblioteca *SDL_mixer*, junto con su gestión y administración básicas. Este esquema encapsula la complejidad del tratamiento del sonido, por lo que en cualquier momento se podría sustituir dicha biblioteca por otra.

Más adelante se muestra un ejemplo concreto en el que se hace uso de este tipo de recursos sonoros. Sin embargo, antes de discutir este ejemplo de integración se planteará el soporte de efectos de sonido, los cuales se podrán mezclar con el tema o *track* principal a la hora de desarrollar un juego. Como se planteará a continuación, la filosofía de diseño es exactamente igual que la planteada en esta sección.

Figura 6.17: La integración de efectos de sonido es esencial para dotar de realismo la parte sonora del videojuego y complementar la parte gráfica.

Soporte de efectos de sonido

Además de llevar a cabo la reproducción del algún tema musical durante la ejecución de un juego, la incorporación de efectos de sonido es esencial para alcanzar un buen grado de inmersión y que, de esta forma, el jugador se sienta como parte del propio juego. Desde un punto de vista técnico, este esquema implica que la biblioteca de desarrollo permita la mezcla de sonidos. En el caso de *SDL_mixer* es posible llevar a cabo dicha integración.

Listado 6.18: Clase TrackManager. Funciones load() y createImpl().

```
// Clase encargada de gestionar recursos del tipo "Track".
// Funcionalidad heredada de Ogre::ResourceManager
// y Ogre::Singleton.
class TrackManager: public Ogre::ResourceManager,
                    public Ogre::Singleton<TrackManager> {
 public:
  TrackManager();
  virtual ~TrackManager();
  // Función de carga genérica.
  virtual TrackPtr load (const Ogre::String& name,
              const Ogre::String& group);
  static TrackManager& getSingleton ();
  static TrackManager* getSingletonPtr ();

 protected:
  // Crea una nueva instancia del recurso.
  Ogre::Resource* createImpl (const Ogre::String& name,
                  Ogre::ResourceHandle handle,
                  const Ogre::String& group,
                  bool isManual,
                  Ogre::ManualResourceLoader* loader,
                  const Ogre::NameValuePairList* createParams);
};

TrackPtr
TrackManager::load
(const Ogre::String& name, const Ogre::String& group)
{
  // Obtención del recurso por nombre...
  TrackPtr trackPtr = getByName(name);

  // Si no ha sido creado, se crea.
  if (trackPtr.isNull())
    trackPtr = create(name, group);

  // Carga explícita del recurso.
  trackPtr->load();

  return trackPtr;
}

// Creación de un nuevo recurso.
Ogre::Resource*
TrackManager::createImpl (const Ogre::String& resource_name,
               Ogre::ResourceHandle handle,
               const Ogre::String& resource_group,
               bool isManual,
               Ogre::ManualResourceLoader* loader,
               const Ogre::NameValuePairList* createParams)
{
  return new Track(this, resource_name, handle,
            resource_group, isManual, loader);
}
```

Como se ha comentado anteriormente, a efectos de implementación, la integración de efectos de sonidos (*FX effects*) sigue el mismo planteamiento que el adoptado para la gestión básica de sonido. Para ello, en primer lugar se han creado las clases **SoundFX** y **SoundFXPtr**, con el objetivo de gestionar y manipular las distintas ins-

tancias de los efectos de sonido, respectivamente. En el siguiente listado de código se muestra la clase *SoundFX*, que se puede entender como una simplificación de la clase *Track*, ya que los efectos de sonido se reproducirán puntualmente, mediante la función miembro *play*, cuando así sea necesario.

La manipulación de efectos de sonido se centraliza en la clase **SoundFXManager**, la cual implementa el patrón *Singleton* y hereda de la clase *Ogre::ResourceManager*. La diferencia más sustancial con respecto al gestor de sonido reside en que el tipo de recurso mantiene un identificador textual distinto y que, en el caso de los efectos de sonido, se lleva a cabo una reserva explícita de 32 canales de audio. Para ello, se hace uso de una función específica de la biblioteca *SDL_mixer*.

Listado 6.19: Clase SoundFX.

```
1  class SoundFX: public Ogre::Resource {
2   public:
3    // Constructor (ver Ogre::Resource).
4    SoundFX(Ogre::ResourceManager* creator,
5        // Igual que en Track...
6        );
7    ~SoundFX();
8
9    int play(int loop = 0); // Reproducción puntual.
10
11   protected:
12    void loadImpl();
13    void unloadImpl();
14    size_t calculateSize() const;
15
16   private:
17    Mix_Chunk* _pSound; // Info sobre el efecto de sonido.
18    Ogre::String _path; // Ruta completa al efecto de sonido.
19    size_t _size;       // Tamaño del efecto (bytes).
20 };
```

Integrando música y efectos

Para llevar a cabo la integración de los aspectos básicos previamente discutidos sobre la gestión de música y efectos de sonido se ha tomado como base el código fuente de la sesión de iluminación del módulo 2, *Programación Gráfica*. En este ejemplo se hacía uso de una clase *MyFrameListener* para llevar a cabo la gestión de los eventos de teclado.

Por una parte, este ejemplo se ha extendido para incluir la reproducción ininterrumpida de un tema musical, es decir, un tema que se estará reproduciendo desde que se inicia la aplicación hasta que ésta finaliza su ejecución. Por otra parte, la reproducción de efectos de sonido adicionales está vinculada a la generación de ciertos eventos de teclado. En concreto, cada vez que el usuario pulsa las teclas '1' ó '2', las cuales están asociadas a dos esquemas diferentes de cálculo del sombreado, la aplicación reproducirá un efecto de sonido puntual. Este efecto de sonido se mezclará de manera adecuada con el *track* principal.

El siguiente listado de código muestra la nueva función miembro introducida en la clase *MyApp* para realizar la carga de recursos asociada a la biblioteca *SDL_mixer*.

Figura 6.19: Captura de pantalla de la ejecución del ejemplo de la sesión de iluminación del módulo 2, *Programación Gráfica*.

Listado 6.20: Clase MyApp. Función initSDL()

```
1 bool
2 MyApp::initSDL () {
3
4     // Inicializando SDL...
```

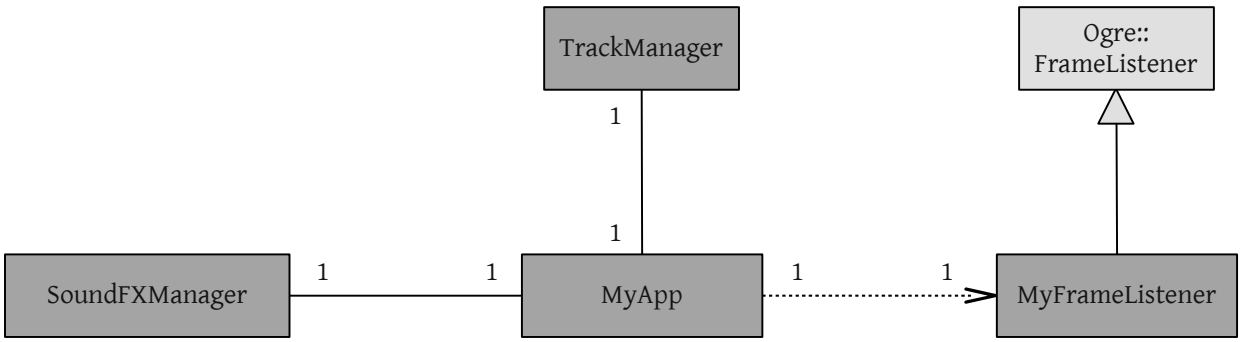


Figura 6.18: Diagrama simplificado de clases de las principales entidades utilizadas para llevar a cabo la integración de música y efectos de sonido.

```
5      if (SDL_Init(SDL_INIT_AUDIO) < 0)
6          return false;
7      // Llamar a SDL_Quit al terminar.
8      atexit(SDL_Quit);
9
10     // Inicializando SDL mixer...
11     if (Mix_OpenAudio(MIX_DEFAULT_FREQUENCY, MIX_DEFAULT_FORMAT,
12                MIX_DEFAULT_CHANNELS, 4096) < 0)
13       return false;
14
15     // Llamar a Mix_CloseAudio al terminar.
16     atexit(Mix_CloseAudio);
17
18     return true;
19
20 }
```

Es importante resaltar que a la hora de arrancar la instancia de la clase *MyApp* mediante la función *start()*, los gestores de sonido y de efectos se instancian. Además, se lleva a cabo la reproducción del track principal.

Listado 6.21: Clase MyApp. Función start()

```
1  // SE OMITE PARTE DEL CÓDIGO FUENTE.
2  int
3  MyApp::start() {
4
5    _root = new Ogre::Root();
6    _pTrackManager = new TrackManager;
7    _pSoundFXManager = new SoundFXManager;
8
9    // Window, cámara y viewport...
10
11   loadResources();
12   createScene();
13   createOverlay();
14
15   // FrameListener...
16
17   // Reproducción del track principal...
18   this->_mainTrack->play();
19
20   _root->startRendering();
21   return 0;
22 }
```

Finalmente, sólo hay que reproducir los eventos de sonido cuando así sea necesario. En este ejemplo, dichos eventos se reproducirán cuando se indique, por parte del usuario, el esquema de cálculo de sombreado mediante las teclas '1' ó '2'. Dicha activación se realiza, por simplificación, en la propia clase *FrameListener*; en concreto, en la función miembro *frameStarted* a la hora de capturar los eventos de teclado.

El esquema planteado para la gestión de sonido mantiene la filosofía de delegar el tratamiento de eventos de teclado, al menos los relativos a la parte sonora, en la clase principal (*MyApp*). Idealmente, si la gestión del bucle de juego se plantea en base a un esquema basado en estados, la reproducción de sonido estaría condicionada por el estado actual. Este planteamiento también es escalable a la hora de integrar nuevos estados de juegos y sus eventos de sonido asociados. La activación de dichos eventos dependerá no sólo del estado actual, sino también de la propia interacción por parte del usuario.

Listado 6.22: Clase MyApp. Función frameStarted()

```
1  // SE OMITE PARTE DEL CÓDIGO FUENTE.
2
3  bool
4  MyFrameListener::frameStarted
5  (const Ogre::FrameEvent& evt) {
6
7    _keyboard->capture();
8    // Captura de las teclas de fecha...
9
10   if(_keyboard->isKeyDown(OIS::KC_1)) {
11     _sceneManager->setShadowTechnique(Ogre::
           SHADOWTYPE_TEXTURE_MODULATIVE);
12     _shadowInfo = "TEXTURE_MODULATIVE";
13     _pMyApp->getSoundFXPtr()->play();  // REPRODUCCIÓN.
14   }
15
16   if(_keyboard->isKeyDown(OIS::KC_2)) {
17     _sceneManager->setShadowTechnique(Ogre::
           SHADOWTYPE_STENCIL_MODULATIVE);
18     _shadowInfo = "STENCIL_MODULATIVE";
19     _pMyApp->getSoundFXPtr()->play();  // REPRODUCCIÓN.
20   }
21
22   // Tratamiento del resto de eventos...
23 }
```

6.3. El sistema de archivos

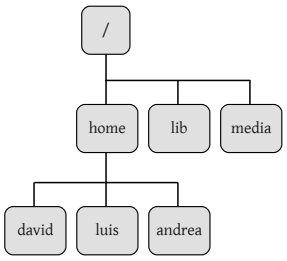


Figura 6.20: Generalmente, los sistemas de archivos mantienen estructuras de árbol o de grafo.

El gestor de recursos hace un uso extensivo del **sistema de archivos**. Típicamente, en los PCs los sistemas de archivos son accesibles mediante llamadas al sistema, proporcionadas por el propio sistema operativo. Sin embargo, en el ámbito de los motores de juegos se suelen plantear esquemas más generales y escalables debido a la diversidad de plataformas existente para la ejecución de un juego.

Esta idea, discutida anteriormente a lo largo del curso, se basa en hacer uso de ***wrappers*** o de capas software adicionales para considerar aspectos clave como el desarrollo multiplataforma. El planteamiento más común consiste en envolver la API del sistema de archivos con una API específica del motor de juegos con un doble objetivo:

- En el desarrollo multiplataforma, esta API específica proporciona el nivel de abstracción necesario para no depender del sistema de archivos y de la plataforma subyacente.

- La API vinculada al sistema de archivos puede no proporcionar toda la funcionalidad necesaria por el motor de juegos. En este caso, la API específica complementa a la nativa.

Por ejemplo, la mayoría de sistemas operativos no proporcionan mecanismos para cargar datos *on the fly* mientras un juego está en ejecución. Por lo tanto, el motor de juegos debería ser capaz de soportar *streaming* de ficheros.

Por otra parte, cada plataforma de juegos maneja un sistema de archivos distinto e incluso necesita gestionar distintos dispositivos de entrada/salida (E/S), como por ejemplo discos *blu-ray* o tarjetas de memoria. Esta variedad se puede ocultar gracias al nivel extra de abstracción proporcionado por la API del propio motor de juegos.

6.3.1. Gestión y tratamiento de archivos

Figura 6.21: Tras acabar con el formato HD-DVD, el Blu-Ray sucederá al formato DVD como principal alternativa en el soporte físico de las consolas de videojuegos de sobremesa.

El sistema de archivos define la organización de los datos y proporciona mecanismos para almacenar, recuperar y actualizar información. Así mismo, también es el encargado de gestionar el espacio disponible en un determinado dispositivo de almacenamiento. Idealmente, el sistema de archivos ha de organizar los datos de una manera eficiente, teniendo en cuenta incluso las características específicas del dispositivo de almacenamiento.

En el ámbito de un motor de juegos, la API vinculada a la gestión del sistema de archivos proporciona la siguiente **funcionalidad** [42]:

- Manipular nombres de archivos y rutas.
- Abrir, cerrar, leer y escribir sobre archivos.
- Listar el contenido de un directorio.
- Manejar peticiones de archivos de manera asíncrona.

Una ruta o ***path*** define la localización de un determinado archivo o directorio y, normalmente, está compuesta por una etiqueta que define el volumen y una serie de componentes separados por algún separador, como por ejemplo la barra invertida. En sistemas UNIX, un ejemplo típico estaría representado por la siguiente ruta:

```
/home/david/apps/firefox/firefox-bin
```

Encapsulación

Una vez más, el principio de encapsulación resulta fundamental para abstraerse de la complejidad asociada al tratamiento del sistema de archivos y del sistema operativo subyacente.

En el caso de las consolas, las rutas suelen seguir un convenio muy similar incluso para referirse a distintos volúmenes. Por ejemplo, PlayStation3™utiliza el prefijo $/dev_bdvd$ para referirse al lector de blu-ray, mientras que el prefijo $/dev_hddx$ permite el acceso a un determinado disco duro.

Las rutas o *paths* pueden ser absolutas o relativas, en función de si están definidas teniendo como referencia el directorio raíz del sistema de archivos u otro directorio, respectivamente. En sistemas UNIX, dos ejemplos típicos serían los siguientes:

```
/usr/share/doc/ogre-doc/api/html/index.html
apps/firefox/firefox-bin
```

El primero de ellos sería una ruta absoluta, ya que se define en base al directorio raíz. Por otra parte, la segunda sería una ruta relativa, ya que se define en relación al directorio $/home/david$.

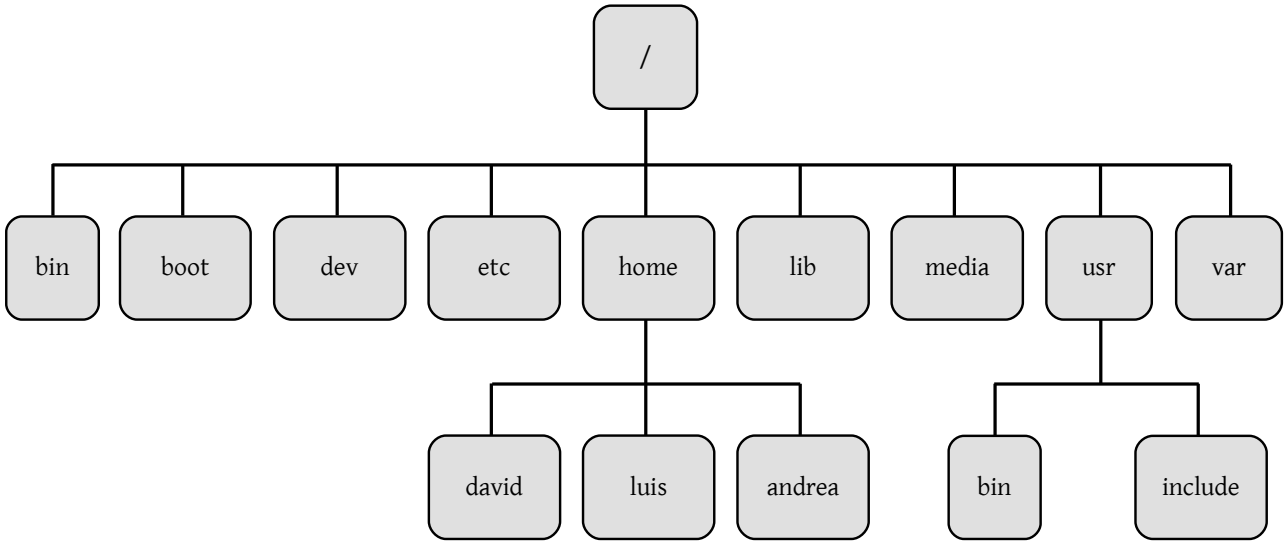


Figura 6.22: Jerarquía típica del sistema de archivos UNIX.

Antes de pasar a discutir aspectos de la gestión de E/S, resulta importante comentar la existencia de **APIs específicas para la gestión de rutas**. Aunque la gestión básica de tratamiento de rutas se puede realizar mediante cadenas de texto, su complejidad hace necesaria, normalmente, la utilización de APIs específicas para abstraer dicha complejidad. La funcionalidad relevante, como por ejemplo la obtención del directorio, nombre y extensión de un archivo, o la conversión entre *paths* absolutos y relativos, se puede encapsular en una API que facilite la interacción con este tipo de componentes.

Instalación

La instalación de la biblioteca *Boost.Filesystem* en sistemas Debian y derivados es trivial mediante cualquier gestor de paquetes. Una búsqueda con *apt-cache search* es todo lo que necesitas para averiguar qué paquete/s hay que instalar.

En el caso del desarrollo de videojuegos multiplataforma, se suele hacer uso de otra API adicional para envolver a las APIs específicas para el tratamiento de archivos en diversos sistemas operativos.

Caso de estudio. La biblioteca Boost.Filesystem

En el contexto de APIs específicas para la gestión de rutas, el uso de la biblioteca *Boost.Filesystem*[5] es un ejemplo representativo de API multiplataforma para el **tratamiento de rutas en C++**. Dicha biblioteca es compatible con el estándar de C++, es portable a diversos sistemas operativos y permite el manejo de errores mediante excepciones.

El siguiente listado de código muestra un programa que realiza un procesamiento recursivo de archivos, mostrando su nombre independientemente del tipo de archivo (regular o directorio). Como se puede apreciar, con un programa trivial es posible llevar a cabo tareas básicas para el tratamiento de rutas.

Listado 6.23: Ejemplo de uso de boost::filesystem.

```
1 #include <iostream>
2 #include <boost/filesystem.hpp>
3
4 using namespace std;
```

[5] www.boost.org/libs/filesystem/

```
 5 using namespace boost::filesystem;
 6
 7 void list_directory (const path& dir, const int& tabs);
 8
 9 int main (int argc, char* argv[]) {
10   if (argc < 2) {
11     cout << "Uso: ./exec/Simple <path>" << endl;
12     return 1;
13   }
14   path p(argv[1]);  // Instancia de clase boost::path.
15
16   if (is_regular_file(p))
17     cout << "   " << p << "  " << file_size(p) << " B" << endl;
18   else if (is_directory(p)) // Listado recursivo.
19     list_directory(p, 0);
20
21   return 0;
22 }
23
24 void
25 print_tabs (const int& tabs) {
26   for (int i = 0; i < tabs; ++i) cout << "\t";
27 }
28
29 void list_directory
30 (const path& p, const int& tabs) {
31   vector<path> paths;
32   // directory iterator para iterar sobre los contenidos del dir.
33   copy(directory_iterator(p), directory_iterator(),
34        back_inserter(paths));
35   sort(paths.begin(), paths.end()); // Se fuerza el orden.
36
37   // Pretty print ;-)
38   for (vector<path>::const_iterator it = paths.begin();
39        it != paths.end();  ++it) {
40     if (is_directory(*it)) {
41       print_tabs(tabs);
42       cout << *it << endl;
43       list_directory(*it, (tabs + 1));
44     }
45     else if (is_regular_file(*it)) {
46       print_tabs(tabs);
47       cout << (*it) << "  " << file_size(*it) << " B" << endl;
48     }
49   }
50 }
```

6.3.2. E/S básica

Para conseguir que la E/S asociada a un motor de juegos sea eficiente, resulta imprescindible plantear un esquema de memoria intermedia que permita gestionar tanto los datos escritos (leídos) como el destino en el que dichos datos se escriben (leen). Típicamente, la solución a este problema está basada en el uso de ***buffers***. Básicamente, un *buffer* representa un puente de comunicación entre el propio programa y un archivo en disco. En lugar de, por ejemplo, realizar la escritura byte a byte en un archivo de texto, se suelen emplear *buffers* intermedios para reducir el número de escrituras en disco y, de este modo, mejorar la eficiencia de un programa.

En esta sección se prestará especial atención a la **biblioteca estándar de C**, ya que es la más comúnmente utilizada para llevar a cabo la gestión de E/S en el desarrollo de videojuegos.

Stream I/O API

La API de E/S con *buffers* que proporciona la biblioteca estándar de C se denomina comúnmente *Stream* I/O API, debido a que proporciona una abstracción que permite gestionar los archivos en disco como flujos de bytes.

El lenguaje de programación C permite manejar dos APIs para gestionar las operaciones más relevantes en relación al contenido de un archivo, como la apertura, lectura, escritura o el cierre. La primera de ellas proporciona E/S con *buffers* mientras que la segunda no.

La diferencia entre ambas reside en que la API con E/S mediante *buffer* gestiona de manera automática el uso de los propios *buffers* sin necesidad de que el programador asuma dicha responsabilidad. Por el contrario, la API de C que no proporciona E/S con *buffers* tiene como consecuencia directa que el programador ha de asignar memoria para dichos *buffers* y gestionarlos. La tabla 6.1 muestra las principales operaciones de dichas APIs.

Listado 6.24: Ejemplo de operación de E/S síncrona

```
1  #include <stdio.h>
2
3  int lectura_sincrona (const char* archivo, char* buffer,
4                size_t tamanyo_buffer, size_t* p_bytes_leidos);
5
6  int main (int argc, const char* argv[]) {
7    char buffer[256];
8    size_t bytes_leidos = 0;
9
10   if (lectura_sincrona("test.txt", buffer, sizeof(buffer), &
         bytes_leidos))
11     printf("%u bytes leidos!\n", bytes_leidos);
12
13   return 0;
14 }
15
16 int lectura_sincrona (const char* archivo, char* buffer,
17               size_t tamanyo_buffer, size_t* p_bytes_leidos) {
18   FILE* manejador = NULL;
19
20   if ((manejador = fopen(archivo, "rb"))) {
21     // Llamada bloqueante en fread,
22     // hasta que se lean todos los datos.
23     size_t bytes_leidos = fread(buffer, 1, tamanyo_buffer,
           manejador);
24
25     // Ignoramos errores...
26     fclose(manejador);
27
28     *p_bytes_leidos = bytes_leidos;
29     return 1;
30   }
31   return -1;
32 }
```

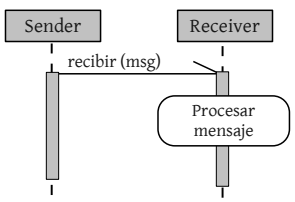


Figura 6.23: El uso de invocaciones síncronas tiene como consecuencia el bloqueo de la entidad que realiza la invocación.

Dependiendo del sistema operativo en cuestión, las invocaciones sobre operaciones de E/S se traducirán en llamadas nativas al sistema operativo, como ocurre en el caso de UNIX y variantes, o en envolturas sobre alguna otra API de más bajo nivel, como es el caso de sistemas Microsoft Windows™. Es posible aprovecharse del hecho de utilizar las **APIs de más bajo nivel**, ya que exponen todos los detalles de implementación del sistema de archivos nativo [42].

Una opción relevante vinculada al uso de *buffers* consiste en gestionarlos para reducir el impacto que la generación de archivos de *log* produce. Por ejemplo, se puede obtener un resultado más eficiente si antes de volcar información sobre un archivo en disco se utiliza un *buffer* intermedio. Así, cuando éste se llena, entonces se vuelca sobre el propio archivo para mejorar el rendimiento de la aplicación. En este contexto, se puede considerar la delegación de esta tarea en un hilo externo al bucle principal del juego.

En relación a lo comentado anteriormente sobre el uso de APIs de más bajo nivel, surge de manera inherente la cuestión relativa al uso de envolturas o *wrappers* sobre la propia biblioteca de E/S, ya sea de bajo nivel y dependiente del sistema operativo o de más alto nivel y vinculada al estándar de C. Comúnmente, los motores de juegos hacen uso de algún tipo de capa software que sirva como envoltura de la API de E/S. Este planteamiento proporcionar tres ventajas importantes [42]:

1. Es posible garantizar que el **comportamiento** del motor en distintas plataformas sea idéntico, incluso cuando la biblioteca nativa de más bajo nivel presente algún tipo de cuestión problemática.
2. Es posible adaptar la API de más alto nivel a las necesidades del motor de juegos. De este modo, se mejora la **mantenibilidad** del mismo ya que sólo se presta atención a los aspectos del sistema de E/S realmente utilizados.
3. Es un esquema **escalable** que permite integrar nueva funcionalidad, como por ejemplo la necesidad de tratar con dispositivos de almacenamiento externos, como unidades de DVD o Blu-Ray.

Finalmente, es muy importante tener en cuenta que las bibliotecas de E/S estándar de C son **síncronas**. En otras palabras, ante una invocación de E/S, el programa se queda *bloqueado* hasta que se atiende por completo la petición. El listado de código anterior muestra un ejemplo de llamada bloqueante mediante la operación *fread()*.

Evidentemente, el esquema síncrono presenta un **inconveniente** muy importante, ya que bloquea al programa mientras la operación de E/S se efectúa, degradando así el rendimiento global de la aplicación. Un posible solución a este problema consiste en adoptar un enfoque asíncrono, tal y como se discute en la siguiente sección.

6.3.3. E/S asíncrona

La E/S asíncrona gira en torno al **concepto de *streaming***, el cual se refiere a la carga de contenido en un segundo plano, es decir, mientras el programa principal está en un primer plano de ejecución. Este esquema posibilita la carga de contenido en tiempo de ejecución, es decir, permite obtener contenido que será utilizado en un futuro inmediato por la propia aplicación, evitando así potenciales cuellos de botella y garantizando que la tasa por segundos del juego no decaiga.

Normalmente, cuando un juego se ejecuta, se hace uso tanto del **disco duro** como de algún tipo de dispositivo de almacenamiento externo, como por ejemplo unidades ópticas, para cargar contenido audiovisual en la memoria principal. Últimamente, uno de los esquemas más utilizados, incluso en consolas de sobremesa, consiste en *instalar* el juego en el disco duro a partir de la copia física del juego, que suele estar en DVD o Blu-Ray. Este proceso consiste en almacenar determinados elementos del juego en el disco duro con el objetivo de agilizar los tiempos de carga en memoria principal.

El uso del disco duro permite la carga de elementos en **segundo plano** mientras el juego continúa su flujo de ejecución normal. Por ejemplo, es perfectamente posible cargar las texturas que se utilizarán en el siguiente nivel mientras el jugador está terminando el nivel actual. De este modo, el usuario no se *desconecta* del juego debido a cuestiones externas.

Para implementar este esquema, la E/S asíncrona resulta fundamental, ya que es una aproximación que permite que el flujo de un programa no se bloquee cuando es necesario llevar a cabo una operación de E/S. En realidad, lo que ocurre es que el flujo del programa continúa mientras se ejecuta la operación de E/S en segundo plano. Cuando ésta finaliza, entonces notifica dicha finalización mediante una **función de retrollamada**.

API E/S con *buffers*

Operación	Signatura
Abrir archivo	FILE *fopen(const char *path, const char *mode);
Cerrar archivo	int fclose(FILE *fp);
Leer archivo	size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
Escribir archivo	size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
Desplazar	int fseek(FILE *stream, long offset, int whence);
Obtener offset	long ftell(FILE *stream);
Lectura línea	char *fgets(char *s, int size, FILE *stream);
Escritura línea	int fputs(const char *s, FILE *stream);
Lectura cadena	int fscanf(FILE *stream, const char *format, ...);
Escritura cadena	int fprintf(FILE *stream, const char *format, ...);
Obtener estado	int fstat(int fd, struct stat *buf);

API E/S sin *buffers*

Operación	Signatura
Abrir archivo	int open(const char *pathname, int flags, mode_t mode);
Cerrar archivo	int close(int fd);
Leer archivo	ssize_t read(int fd, void *buf, size_t count);
Escribir archivo	ssize_t write(int fd, const void *buf, size_t count);
Desplazar	off_t lseek(int fd, off_t offset, int whence);
Obtener offset	off_t tell(int fd);
Lectura línea	No disponible
Escritura línea	No disponible
Lectura cadena	No disponible
Escritura cadena	No disponible
Obtener estado	int stat(const char *path, struct stat *buf);

Cuadro 6.1: Resumen de las principales operaciones de las APIs de C para la gestión de E/S (con y sin *buffers*).

La figura 6.24 muestra de manera gráfica cómo dos agentes se comunican de manera asíncrona mediante un objeto de retrollamada. Básicamente, el *agente notificador* envía un mensaje al *agente receptor* de manera asíncrona. Esta invocación tiene dos parámetros: i) el propio mensaje m y ii) un objeto de retrollamada cb. Dicho objeto será usado por el *agente receptor* cuando éste termine de procesar el mensaje. Mientras tanto, el *agente notificador* continuará su flujo normal de ejecución, sin necesidad de bloquearse por el envío del mensaje.

Callbacks

Las funciones de retrollamada se suelen denominar comúnmente *callbacks*. Este concepto no sólo se usa en el ámbito de la E/S asíncrona, sino también en el campo de los sistemas distribuidos y los *middlewares* de comunicaciones.

La mayoría de las bibliotecas de E/S asíncrona existentes permiten que el programa principal espera una cierta cantidad de tiempo antes de que una operación de E/S se complete. Este planteamiento puede ser útil en situaciones en las que los datos de dicha operación se necesitan para continuar con el flujo normal de trabajo. Otras APIs pueden proporcionar incluso una estimación del tiempo que tardará en completarse una operación de E/S, con el objetivo de que el programador cuente con la mayor cantidad posible de información. Así mismo, también es posible encontrarse con operaciones que asignen *deadlines* sobre las propias peticiones, con el objetivo de plantear un esquema basado en prioridades. Si el *deadline* se cumple, entonces también suele ser posible asignar el código a ejecutar para contemplar ese caso en particular.

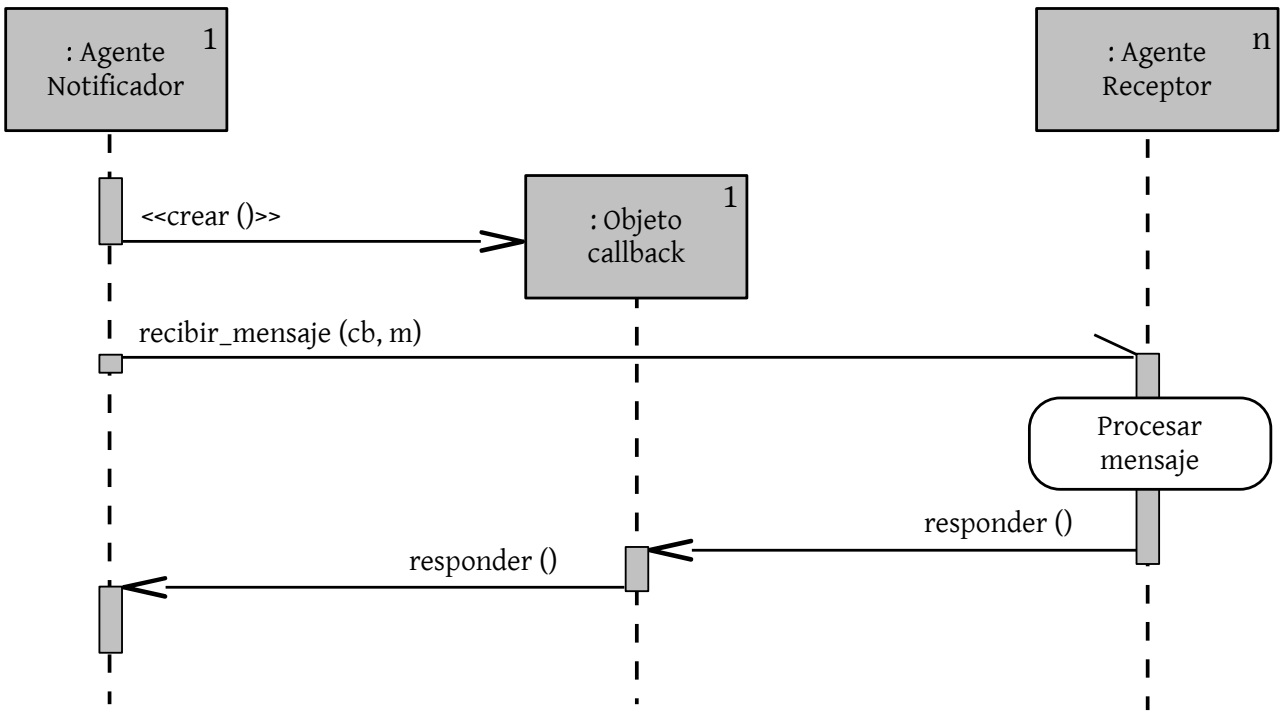


Figura 6.24: Esquema gráfico representativo de una comunicación asíncrona basada en objetos de retrollamada.

Desde un punto de vista general, la E/S asíncrona se suele implementar mediante **hilos auxiliares** encargados de atender las peticiones de E/S. De este modo, el hilo principal ejecuta funciones que tendrán como consecuencia peticiones que serán encoladas para atenderlas posteriormente. Este hilo principal retornará inmediatamente después de llevar a cabo la invocación.

El hilo de E/S obtendrá la siguiente petición de la cola y la atenderá utilizando funciones de E/S bloqueantes, como la función *fread* discutida en el anterior listado de código. Cuando se completa una petición, entonces se invoca al objeto o a la función de retrollamada proporcionada anteriormente por el hilo principal (justo cuando se realizó la petición inicial). Este objeto o función notificará la finalización de la petición.

En caso de que el hilo principal tenga que esperar a que la petición de E/S se complete antes de continuar su ejecución será necesario proporcionar algún tipo de mecanismo para garantizar dicho bloqueo. Normalmente, se suele hacer uso de semáforos [89] (ver figura 6.25), vinculando un semáforo a cada una de las peticiones. De este modo, el hilo principal puede hacer uso de *wait* hasta que el hilo de E/S haga uso de *signal*, posibilitando así que se reanude el flujo de ejecución.

6.3.4. Caso de estudio. La biblioteca Boost.Asio C++

Boost.Asio[6] es una biblioteca multiplataforma desarrollada en C++ con el objetivo de dar soporte a operaciones de red y de E/S de bajo nivel a través de un **modelo asíncrono** consistente y bajo un enfoque moderno. *Asio* se enmarca dentro del proyecto *Boost*.

El desarrollo de esta biblioteca se justifica por la necesidad de interacción entre programas, ya sea mediante ficheros, redes o mediante la propia consola, que no pueden quedarse a la espera ante operaciones de E/S cuyo tiempo de ejecución sea elevado. En el caso del desarrollo de videojuegos, una posible aplicación de este enfoque, tal y como se ha introducido anteriormente, sería la carga en **segundo plano** de recursos que serán utilizados en el futuro inmediato. La situación típica en este contexto sería la carga de los datos del siguiente nivel de un determinado juego.

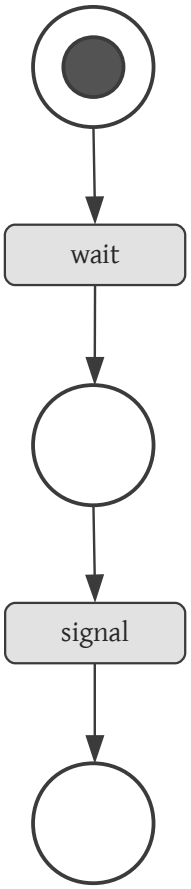


Figura 6.25: Esquema general de uso de un semáforo para controlar el acceso a determinadas secciones de código.

Figura 6.26: Las bibliotecas del proyecto Boost representan, generalmente, una excelente alternativa para solucionar un determinado problema.

Según el propio desarrollador de la biblioteca[7], *Asio* proporciona las herramientas necesarias para manejar este tipo de problemática sin la necesidad de utilizar, por parte del programador, modelos de concurrencia basados en el uso de hilos y mecanismos de exclusión mutua. Aunque la biblioteca fue inicialmente concebida para la problemática asociada a las redes de comunicaciones, ésta es perfectamente aplicable para llevar a cabo una E/S asíncrona asociada a descriptores de ficheros o incluso a puertos serie.

Los principales **objetivos de diseño** de *Boost.Asio* son los siguientes:

- **Portabilidad**, gracias a que está soportada en un amplio rango de sistemas operativos, proporcionando un comportamiento consistente en los mismos.
- **Escalabilidad**, debido a que facilita el desarrollo de aplicaciones de red que escalen a un gran número de conexiones concurrentes. En principio, la implementación de la biblioteca en un determinado sistema operativo se beneficia del soporte nativo de éste para garantizar esta propiedad.
- **Eficiencia**, gracias a que la biblioteca soporta aspectos relevantes, como por ejemplo la reducción de las copias de datos.
- **Reutilización**, debido a que *Asio* se basa en modelos y conceptos ya establecidos, como la API BSD *Socket*.
- **Facilidad de uso**, ya que la biblioteca está orientada a proporcionar un *kit* de herramientas, en lugar de basarse en el modelo *framework*.

La instalación de la biblioteca *Boost.Asio* en sistemas Debian y derivados es trivial mediante los siguientes comandos:

```
$ sudo apt-get update
$ sudo apt-get install libasio-dev
```

A continuación se plantean algunos ejemplos ya implementados con el objetivo de ilustrar al lector a la hora de plantear un esquema basado en el asincronismo para, por ejemplo, llevar a cabo operaciones de E/S asíncrona o cargar recursos en *segundo plano*[8]. El siguiente listado muestra un programa de uso básico de la biblioteca *Asio*, en el que se muestra la consecuencia directa de una llamada bloqueante.

[6] http://think-async.com/Asio/

[7] http://www.boost.org/doc/libs/1_48_0/doc/html/boost_asio.html

[8] La noción de segundo plano en este contexto está vinculada a independizar ciertas tareas de la ejecución del flujo principal de un programa.

Listado 6.25: Biblioteca Asio. Espera explícita.

```
1  #include <iostream>
2  #include <boost/asio.hpp>
3  #include <boost/date_time/posix_time/posix_time.hpp>
4
5  int main () {
6    // Todo programa que haga uso de asio ha de instanciar
7    // un objeto del tipo io service para manejar la E/S.
8    boost::asio::io_service io;
9
10   // Instancia de un timer (3 seg.)
11   // El primer argumento siempre es un io service.
12   boost::asio::deadline_timer t(io, boost::posix_time::seconds(3));
13
14   // Espera explícita.
15   t.wait();
16
17   std::cout << "Hola Mundo!" << std::endl;
18
19   return 0;
20 }
```

Para compilar, enlazar y generar el archivo ejecutable, simplemente es necesario ejecutar las siguientes instrucciones:

```
$ g++ Simple.cpp -o Simple -lboost_system -lboost_date_time
$ ./Simple
```

In the meantime...

Recuerde que mientras se lleva a cabo una espera asíncrona es posible continuar ejecutando operaciones en el hilo principal. Aproveche este esquema para obtener un mejor rendimiento en sistemas con más de un procesador.

En determinados contextos resulta muy deseable llevar a cabo una **espera asíncrona**, es decir, continuar ejecutando instrucciones mientras en otro nivel de ejecución se realizan otras tareas. Si se utiliza la biblioteca *Boost.Asio*, es posible definir manejadores de código asociados a funciones de retrollamada que se ejecuten mientras el programa continúa la ejecución de su flujo principal. *Asio* también proporciona mecanismos para que el programa no termine mientras haya tareas por finalizar.

El siguiente listado de código muestra cómo el ejemplo anterior se puede modificar para llevar a cabo una espera asíncrona, asociando en este caso un temporizador que controla la ejecución de una función de retrollamada.

Listado 6.26: Biblioteca Asio. Espera asíncrona.

```
1  #include <iostream>
2  #include <boost/asio.hpp>
3  #include <boost/date_time/posix_time/posix_time.hpp>
4
5  // Función de retrollamada.
6  void print (const boost::system::error_code& e) {
7    std::cout << "Hola Mundo!" << std::endl;
8  }
9
10 int main () {
11   boost::asio::io_service io;
12
13   boost::asio::deadline_timer t(io, boost::posix_time::seconds(3));
14   // Espera asíncrona.
15   t.async_wait(&print);
16
17   std::cout << "Esperando a print()..." << std::endl;
18   // Pasarán casi 3 seg. hasta que print se ejecute...
19
20   // io.run() para garantizar que los manejadores se llamen desde
21   // hilos que llamen a run().
22   // run() se ejecutará mientras haya cosas por hacer,
23   // en este caso la espera asíncrona a print.
24   io.run();
```

```
25
26   return 0;
27 }
```

La ejecución de este programa generará la siguiente salida:

```
$ Esperando a print()...
$ Hola Mundo!
```

Sin embargo, pasarán casi 3 segundos entre la impresión de una secuencia y otra, ya que inmediatamente después de la llamada a la espera asíncrona (línea 15) se ejecutará la primera secuencia de impresión (línea 17), mientras que la ejecución de la función de retrollamada *print()* (líneas 6-8) se demorará 3 segundos.

Otra opción imprescindible a la hora de gestionar estos manejadores reside en la posibilidad de realizar un **paso de parámetros**, en función del dominio de la aplicación que se esté desarrollando. El siguiente listado de código muestra cómo llevar a cabo dicha tarea.

Listado 6.27: Biblioteca Asio. Espera asíncrona y paso de parámetros.

```
1  #define THRESHOLD 5
2
3  // Función de retrollamada con paso de parámetros.
4  void count (const boost::system::error_code& e,
5           boost::asio::deadline_timer* t, int* counter) {
6    if (*counter < THRESHOLD) {
7      std::cout << "Contador... " << *counter << std::endl;
8      ++(*counter);
9
10     // El timer se prolonga un segundo...
11     t->expires_at(t->expires_at() + boost::posix_time::seconds(1));
12     // Llamada asíncrona con paso de argumentos.
13     t->async_wait(boost::bind
14           (count, boost::asio::placeholders::error, t, counter));
15   }
16 }
17
18 int main () {
19    int counter = 0;
20   boost::asio::deadline_timer t(io, boost::posix_time::seconds(1));
21   // Llamada inicial.
22   t.async_wait(boost::bind
23             (count, boost::asio::placeholders::error,
24          &t, &counter));
25   // ...
26 }
```

Como se puede apreciar, las llamadas a la función *async_wait()* varían con respecto a otros ejemplos, ya que se indica de manera explícita los argumentos relevantes para la función de retrollamada. En este caso, dichos argumentos son la propia función de retrollamada, para realizar llamadas recursivas, el *timer* para poder prolongar en un segundo su duración en cada llamada, y una variable entera que se irá incrementando en cada llamada.

Flexibilidad en Asio

La biblioteca Boost.Asio es muy flexible y posibilita la llamada asíncrona a una función que acepta un número arbitrario de parámetros. Éstos pueden ser variables, punteros, o funciones, entre otros.

La biblioteca *Boost.Asio* también permite encapsular las funciones de retrollamada que se ejecutan de manera asíncrona como **funciones miembro** de una clase. Este esquema mejora el diseño de la aplicación y permite que el desarrollador se abstraiga de la implementación interna de la clase. El siguiente listado de código muestra una posible modificación del ejemplo anterior mediante la definición de una clase *Counter*, de manera que la función *count* pasa a ser una función miembro de dicha clase.

Listado 6.28: Biblioteca Asio. Espera asíncrona y clases.

```
1  class Counter {
2  public:
3    Counter (boost::asio::io_service& io)
4      : _timer(io, boost::posix_time::seconds(1)), _count(0) {
5      _timer.async_wait(boost::bind(&Counter::count, this));
6    }
7    ~Counter () { cout << "Valor final: " << _count << endl; }
8
9    void count () {
10     if (_count < 5) {
11       std::cout << _count++ << std::endl;
12       _timer.expires_at(_timer.expires_at() +
13             boost::posix_time::seconds(1));
14       // Manejo de funciones miembro.
15       _timer.async_wait(boost::bind(&Counter::count, this));
16     }
17   }
18 private:
19   boost::asio::deadline_timer _timer;
20   int _count;
21 };
22
23 int main () {
24   boost::asio::io_service io;
25   Counter c(io); // Instancia de la clase Counter.
26   io.run();
27   return 0;
28 }
```

En el contexto de carga de contenido en un juego de manera independiente al hilo de control principal, el ejemplo anterior no serviría debido a que presenta dos importantes limitaciones. Por una parte, la respuesta de la función de retrollamada no es controlable si dicha retrollamada tarda mucho tiempo en completarse. Por otra parte, este planteamiento no es escalable a sistemas multiproceso.

Para ello, una posible solución consiste en hacer uso de un ***pool* de hilos** que interactúen con *io_service::run()*. Sin embargo, este esquema plantea un nuevo problema: la necesidad de sincronizar el acceso concurrente de varios manejadores a los recursos compartidos, como por ejemplo una variable miembro de una clase.

El siguiente listado de código extiende la definición de la clase *Counter* para manejar de manera concurrente dos *timers* que irán incrementado la variable miembro *_count*.

Listado 6.29: Biblioteca Asio. Espera asíncrona y clases.

```
1  class Counter {
2  public:
3    Counter (boost::asio::io_service& io)
4      : _strand(io), // Para garantizar la exclusión mutua.
5        _timer1(io, boost::posix_time::seconds(1)),
6        _timer2(io, boost::posix_time::seconds(1)),
7        _count(0) {
8      // Los manejadores se 'envuelven' por strand para
9      // que no se ejecuten de manera concurrente.
10     _timer1.async_wait(_strand.wrap
11                 (boost::bind(&Counter::count1, this)));
12     _timer2.async_wait(_strand.wrap
13                 (boost::bind(&Counter::count2, this)));
14   }
15
16   // count1 y count2 nunca se ejecutarán en paralelo.
17   void count1() {
18     if (_count < 10) {
19       // IDEM que en el ejemplo anterior.
```

```
20        _timer1.async_wait(_strand.wrap
21                (boost::bind(&Counter::count1, this)));
22      }
23    }
24
25    // IDEM que count1 pero sobre timer2 y count2
26    void count2() { /* src */ }
27
28  private:
29    boost::asio::strand _strand;
30    boost::asio::deadline_timer _timer1, _timer2;
31    int _count;
32  };
33
34  int main() {
35    boost::asio::io_service io;
36    // run() se llamará desde dos threads (principal y boost)
37    Counter c(io);
38    boost::thread t(boost::bind(&boost::asio::io_service::run, &io));
39    io.run();
40    t.join();
41
42    return 0;
43  }
```

En este caso, la biblioteca proporciona *wrappers* para *envolver* las funciones de retrollamada *count1* y *count2*, con el objetivo de que, internamente, *Asio* controle que dichos manejadores no se ejecutan de manera concurrente. En el ejemplo propuesto, la variable miembro *_count* no se actualizará, simultáneamente, por dichas funciones miembro.

También resulta interesante destacar la instanciación de un hilo adicional a través de la clase *boost::thread*, de manera que la función *io_service::run()* se llame tanto desde este hilo como desde el principal. La filosofía de trabajo se mantiene, es decir, los hilos se seguirán ejecutando mientras haya trabajo pendiente. En concreto, el hilo en segundo plano no finalizará hasta que todas las operaciones asíncronas hayan terminado su ejecución.

6.3.5. Consideraciones finales

En el siguiente capítulo se discutirá otro enfoque basado en la concurrencia mediante hilos en C++ para llevar a cabo tareas como por ejemplo la carga de contenido, en segundo plano, de un juego. En dicho capítulo se utilizará como soporte la biblioteca de utilidades del *middleware* ZeroC ICE, el cual también se estudiará en el módulo 3, *Técnicas Avanzadas de Desarrollo*.

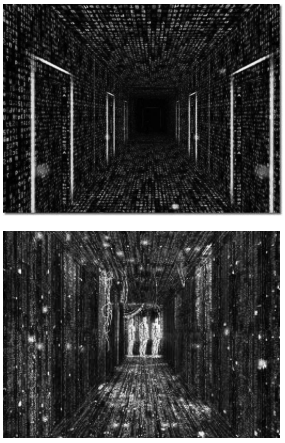


Figura 6.27: Serialización de escenarios de Matrix... ;-)

6.4. Importador de datos de intercambio

En esta sección se justifica la necesidad de plantear esquemas de datos para importar contenido desde entornos de creación de contenido 3D, como por ejemplo Blender. Aunque existen diversos estándares para la definición de datos multimedia, en la práctica cada aplicación gráfica interactiva tiene necesidades específicas que han de cubrirse con un **software particular** para importar dichos datos.

Por otra parte, el **formato de los datos** es otro aspecto relevante, existiendo distintas aproximaciones para codificar la información multimedia a importar. Uno de los esquemas más utilizados consiste en hacer uso del metalenguaje XML (eXtensible Markup Language), por lo que en este capítulo se discutirá esta opción en detalle.

XML mantiene un alto nivel semántico, está bien soportado y estandarizado y permite asociar estructuras de árbol bien definidas para encapsular la información relevante. Además, posibilita que el contenido a importar sea legible por los propios programadores.

Es importante resaltar que este capítulo está centrado en la parte de importación de datos hacia el motor de juegos, por lo que el proceso de exportación no se discutirá a continuación (si se hará, no obstante, en el módulo 2 de *Programación Gráfica*).

6.4.1. Formatos de intercambio

La **necesidad de intercambiar información** es una constante no sólo entre los distintos componentes de un motor de juegos, sino también entre entornos de creación de contenido 3D y el propio motor. El modelado y la animación de personajes y la creación de escenarios se realiza mediante este tipo de *suites* de creación 3D para, posteriormente, integrarlos en el juego. Este planteamiento tiene como consecuencia la necesidad de abordar un doble proceso (ver figura 6.28) para llevar a cabo la interacción entre las partes implicadas:

1. **Exportación**, con el objetivo de obtener una representación de los datos 3D.
2. **Importación**, con el objetivo de acceder a dichos datos desde el motor de juegos o desde el propio juego.

En el caso particular de Ogre3D, existen diversas herramientas[9] que posibilitan la exportación de datos a partir de un determinado entorno de creación de contenido 3D, como por ejemplo Blender, 3DS Max, Maya o Softimage, entre otros. El proceso inverso también se puede ejecutar haciendo uso de herramientas como *OgreXmlConverter*[10], que permiten la conversión entre archivos *.mesh* y *.skeleton* a ficheros XML y viceversa.

El proceso de importación en Ogre3D está directamente relacionado con el formato XML a través de una serie de estructuras bien definidas para manejar escenas, mallas o esqueletos, entre otros.

Aunque el uso del metalenguaje XML es una de las opciones más extendidas, a continuación se discute brevemente el uso de otras alternativas.

Archivos binarios

El mismo planteamiento discutido que se usa para llevar a cabo el almacenamiento de las variables de configuración de un motor de juegos, utilizando un formato binario, se puede aplicar para almacenar información multimedia con el objetivo de importarla posteriormente.

Este esquema basado en un formato binario es muy eficiente y permite hacer uso de técnicas de **serialización de objetos**, que a su vez incrementan la portabilidad y la sencillez de los formatos binarios. En el caso de hacer uso de orientación a objetos, la funcionalidad necesaria para serializar y *de-serializar* objetos se puede encapsular en la propia definición de clase.

Es importante destacar que la serialización de objetos puede generar información en formato XML, es decir, no tiene por qué estar ligada a un formato binario.

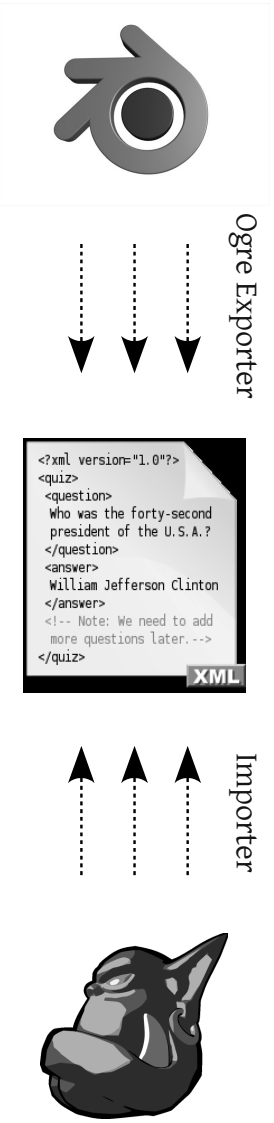


Figura 6.28: Procesos de importación y exportación de datos 3D haciendo uso de documentos XML.

[9] http://www.ogre3d.org/tikiwiki/OGRE+Exporters
[10] http://www.ogre3d.org/tikiwiki/OgreXmlConverter

Archivos en texto plano

El uso de archivos en texto plano es otra posible alternativa a la hora de importar datos. Sin embargo, esta aproximación tiene dos importantes desventajas: i) resulta menos eficiente que el uso de un formato binario e ii) implica la utilización de un procesador de texto específico.

En el caso de utilizar un formato no estandarizado, es necesario explicitar los separadores o *tags* existentes entre los distintos campos del archivo en texto plano. Por ejemplo, se podría pensar en separadores como los dos puntos, el punto y coma y el retorno de carro para delimitar los campos de una determinada entidad y una entidad de la siguiente, respectivamente.

XML

EXtensible Markup Language (XML) se suele definir como un **metalenguaje**, es decir, como un lenguaje que permite la definición de otros lenguajes. En el caso particular de este capítulo sobre datos de intercambio, XML se podría utilizar para definir un lenguaje propio que facilite la importación de datos 3D al motor de juegos o a un juego en particular.

Actualmente, XML es un formato muy popular debido a los siguientes motivos:

- Es un estándar.
- Está muy bien soportado, tanto a nivel de programación como a nivel de usuario.
- Es legible.
- Tiene un soporte excelente para estructuras de datos jerárquicas; en concreto, de tipo arbóreas. Esta propiedad es especialmente relevante en el dominio de los videojuegos.

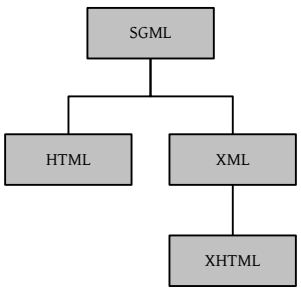


Figura 6.29: Visión conceptual de la relación de XML con otros lenguajes.

Sin embargo, no todo son ventajas. El proceso de *parseado* o ***parsing*** es relativamente lento. Esto implica que algunos motores hagan uso de formatos binarios propietarios, los cuales son más rápidos de *parsear* y mucho más compactos que los archivos XML, reduciendo así los tiempos de importación y de carga.

El *parser* Xerces-C++

Como se ha comentado anteriormente, una de los motivos por los que XML está tan extendido es su amplio soporte a nivel de programación. En otras palabras, prácticamente la mayoría de lenguajes de programación proporcionan bibliotecas, APIs y herramientas para procesar y generar contenidos en formato XML.

Figura 6.30: Logo principal del proyecto Apache.

En el caso del lenguaje C++, estándar *de facto* en la industria del videojuego, el *parser* Xerces-C++[11] es una de las alternativas más completas para llevar a cabo el procesamiento de ficheros XML. En concreto, esta herramienta forma parte del proyecto *Apache* XML[12], el cual gestiona un número relevante de subproyectos vinculados al estándar XML.

[11] `http://xerces.apache.org/xerces-c/`
[12] `http://xml.apache.org/`

Xerces-C++ está escrito en un subconjunto portable de C++ y tiene como objetivo facilitar la lectura y escritura de archivos XML. Desde un punto de vista técnico, Xerces-C++ proporcionar una **biblioteca** con funcionalidad para *parsear*, generar, manipular y validar documentos XML utilizando las APIs DOM, SAX (Simple API for XML) y SAX2. Estas APIs son las más populares para manipular documentos XML y difieren, sin competir entre sí, en aspectos como el origen, alcance o estilo de programación.

Por ejemplo, mientras DOM (Document Object Model) está siendo desarrollada por el consorcio W3, SAX (Simple API for XML)) no lo está. Sin embargo, la mayor diferencia reside en el modelo de programación, ya que SAX presenta el documento XML como una *cadena de eventos serializada*, mientras que DOM lo trata a través de una estructura de árbol. La principal desventaja del enfoque planteado en SAX es que no permite un acceso aleatorio a los elementos del documento. No obstante, este enfoque posibilita que el desarrollador no se preocupe de información irrelevante, reduciendo así el tamaño en memoria necesario por el programa.

Desde un punto de vista general, se recomienda el uso de DOM en proyectos en los que se vayan a integrar distintos componentes de código y las necesidades funcionales sean más exigentes. Por el contrario, SAX podría ser más adecuado para flujos de trabajo más acotados y definidos.

La **instalación** de Xerces-C++, incluyendo documentación y ejemplos de referencia, en sistemas Debian y derivados es trivial mediante los siguientes comandos:

```
$ sudo apt-get update
$ sudo apt-get install libxerces-c-dev
$ sudo apt-get install libxerces-c-doc
$ sudo apt-get install libxerces-samples
```

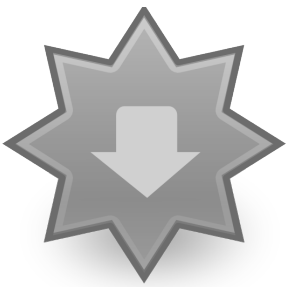


Figura 6.31: La instalación de paquetes en sistemas Debian también se puede realizar a través de gestores de más alto nivel, como por ejemplo *Synaptic*.

6.4.2. Creación de un importador

Justificación y estructura XML

En esta sección se discute el diseño e implementación de un importador específico para un juego muy sencillo titulado ***NoEscapeDemo***, el cual hará uso de la biblioteca Xerces-C++ para llevar a cabo la manipulación de documentos XML.

Básicamente, el juego consiste en interactuar con una serie de *wumpus* o fantasmas que aparecen de manera automática en determinados puntos de un escenario y que desaparecen en otros. La interacción por parte del usuario consiste en modificar el comportamiento de dichos fantasmas cambiando, por ejemplo, el sentido de navegación de los mismos.

La figura 6.32 muestra, desde un punto de vista abstracto, la representación del escenario en el que los fantasmas habitan. Como se puede apreciar, la representación interna de dicho escenario es un grafo, el cual define la navegabilidad de un punto en el espacio 3D a otro. En el grafo, algunos de los nodos están etiquetados con los identificadores **S** o **D**, representando puntos de aparición y desaparición de fantasmas.

El contenido gráfico de esta sencilla *demo* está compuesto de los siguientes elementos:

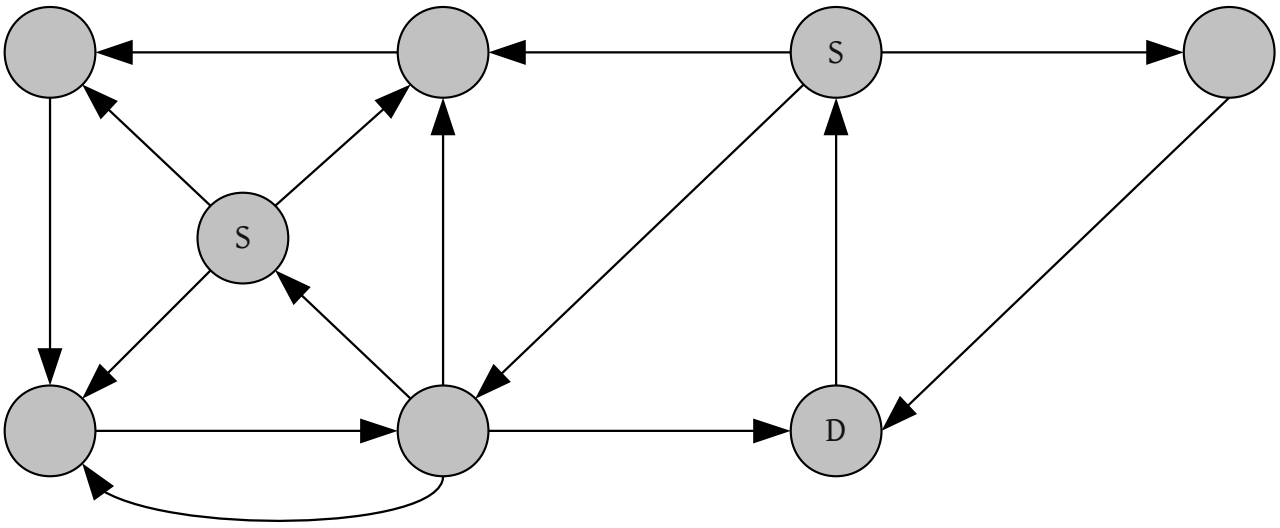


Figura 6.32: Representación interna bidimensional en forma de grafo del escenario de *NoEscapeDemo*. Los nodos de tipo **S** (*spawn*) representan puntos de nacimiento, mientras que los nodos de tipo **D** (*drain*) representan sumideros, es decir, puntos en los que los fantasmas desaparecen.

- El propio **escenario** tridimensional, considerando los puntos o nodos de generación y destrucción de fantasmas.
- Los propios **fantasmas**.
- Una o más **cámaras virtuales**, que servirán para visualizar el juego. Cada cámara tendrá asociado un camino o *path*, compuesto a su vez por una serie de puntos clave que indican la situación (posición y rotación) de la cámara en cada momento.

El nodo raíz

La etiqueta <*data*> es el nodo raíz del documento XML. Sus posibles nodos hijo están asociados con las etiquetas <*graph*> y <*camera*>.

En principio, la información del escenario y de las cámaras virtuales será importada al juego, haciendo uso del importador cuyo diseño se discute a continuación. Sin embargo, antes se muestra un posible ejemplo de la representación de estos mediante el formato definido para la *demo* en cuestión.

En concreto, el siguiente listado muestra la parte de información asociada a la estructura de grafo que conforma el escenario. Como se puede apreciar, la estructura ***graph*** está compuesta por una serie de vértices (*vertex*) y de arcos (*edge*).

Por una parte, en los **vértices** del grafo se incluye su posición en el espacio 3D mediante las etiquetas <*x*>, <*y*> y <*z*>. Además, cada vértice tiene como atributo un índice, que lo identifica unívocamente, y un tipo, el cual puede ser *spawn* (punto de generación) o *drain* (punto de desaparición), respectivamente. Por otra parte, los **arcos** permiten definir la estructura concreta del grafo a importar mediante la etiqueta <*vertex*>.

Vértices y arcos

En el formato definido, los vértices se identifican a través del atributo *index*. Estos IDs se utilizan en la etiqueta <*edge*> para especificar los dos vértices que conforman un arco.

En caso de que sea necesario incluir más contenido, simplemente habrá que extender el formato definido. XML facilita enormemente la escalabilidad gracias a su esquema basado en el uso de etiquetas y a su estructura jerárquica.

Listado 6.30: Ejemplo de fichero XML usado para exportar e importar contenido asociado al grafo del escenario.

```
<?xml version='1.0' encoding='UTF-8'?>
<data>

  <graph>

    <vertex index="1" type="spawn">
      <x>1.5</x> <y>2.5</y> <z>-3</z>
    </vertex>
    <!-- More vertexes... -->
    <vertex index="4" type="drain">
      <x>1.5</x> <y>2.5</y> <z>-3</z>
    </vertex>

    <edge>
      <vertex>1</vertex> <vertex>2</vertex>
    </edge>
    <edge>
      <vertex>2</vertex> <vertex>4</vertex>
    </edge>
    <!-- More edges... -->

  </graph>

  <!-- Definition of virtual cameras -->
</data>
```

El siguiente listado muestra la otra parte de la estructura del documento XML utilizado para importar contenido. Éste contiene la información relativa a las **cámaras virtuales**. Como se puede apreciar, cada cámara tiene asociado un camino, definido con la etiqueta <*path*>, el cual consiste en una serie de puntos o *frames* clave que determinan la animación de la cámara.

Cada punto clave del camino de una cámara contiene la posición de la misma en el espacio 3D (etiqueta <*frame*>) y la rotación asociada, expresada mediante un cuaternión (etiqueta <*rotation*>).

Lógica de dominio

La figura 6.33 muestra el diagrama de las principales clases vinculadas al importador de datos. Como se puede apreciar, las entidades más relevantes que forman parte del documento XML se han modelado como clases con el objetivo de facilitar no sólo la obtención de los datos sino también su integración en el despliegue final con Ogre3D.

La clase que centraliza la lógica de dominio es la clase **Scene**, la cual mantiene una relación de asociación con las clases *Graph* y *Camera*. Recuerde que el contenido a importar más relevante sobre el escenario era la estructura de grafo del mismo y la información de las cámaras virtuales. El listado de código 6.32 muestra la declaración de la clase *Scene*.

La clase *Scene*

La encapsulación de los datos importados de un documento XML se centraliza en la clase *Scene*, la cual mantiene como estado un puntero a un objeto de tipo *Graph* y una estructura con punteros a objetos de tipo *Camera*.

Listado 6.32: Clase Scene.

```
#include <vector>
#include <Camera.h>
```

Listado 6.31: Ejemplo de fichero XML usado para exportar e importar contenido asociado a las cámaras virtuales.

```
<?xml version='1.0' encoding='UTF-8'?>
<data>
  <!-- Graph definition here-->

  <camera index="1" fps="25">

    <path>

      <frame index="1">
    <position>
      <x>1.5</x> <y>2.5</y> <z>-3</z>
    </position>
    <rotation>
      <x>0.17</x> <y>0.33</y> <z>0.33</z> <w>0.92</w>
    </rotation>
      </frame>

      <frame index="2">
    <position>
      <x>2.5</x> <y>2.5</y> <z>-3</z>
    </position>
    <rotation>
      <x>0.17</x> <y>0.33</y> <z>0.33</z> <w>0.92</w>
    </rotation>
      </frame>

      <!-- More frames here... -->

    </path>

  </camera>

  <!-- More cameras here... -->
</data>
```

```
#include <Node.h>
#include <Graph.h>

class Scene
{
 public:
  Scene ();
  ~Scene ();

  void addCamera (Camera* camera);
  Graph* getGraph () { return _graph;}
  std::vector<Camera*> getCameras () { return _cameras; }

 private:
  Graph *_graph;
  std::vector<Camera*> _cameras;
};
```

Por otra parte, la clase **Graph** mantiene la lógica de gestión básica para implementar una estructura de tipo grafo mediante listas de adyacencia. El siguiente listado de código muestra dicha clase e integra una función miembro para obtener la lista de vértices o nodos adyacentes a partir del identificador de uno de ellos (línea 17).

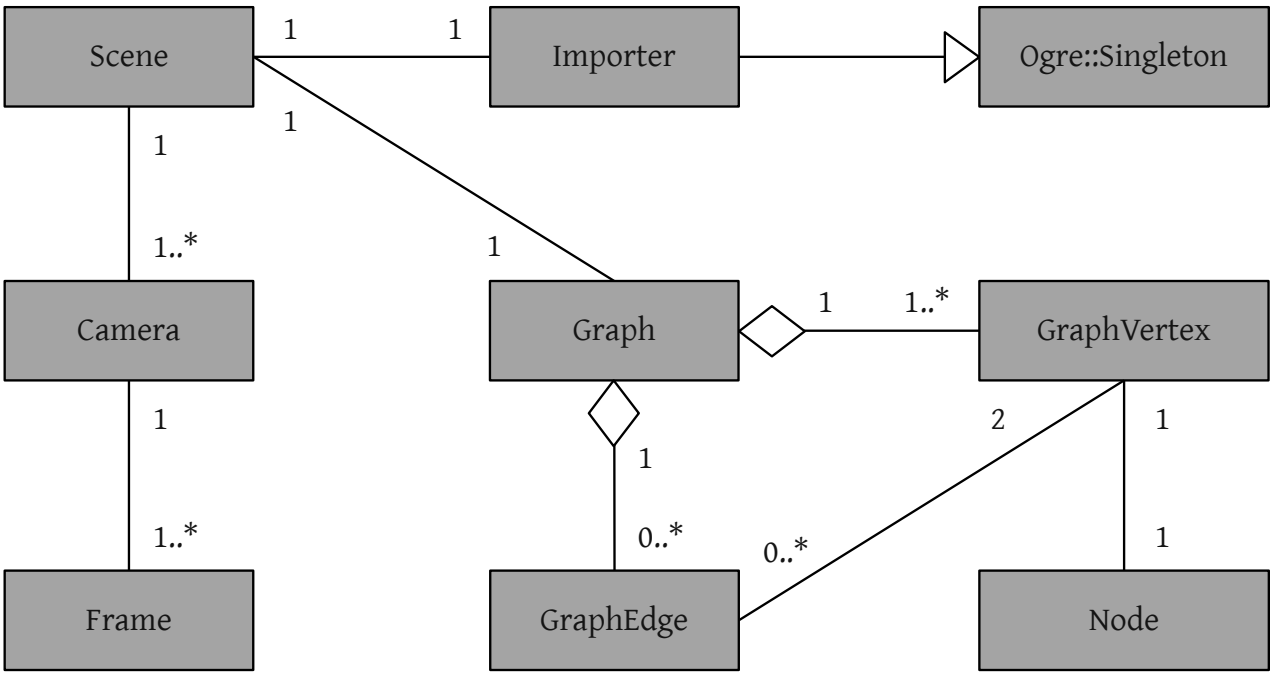


Figura 6.33: Diagrama de clases de las entidades más relevantes del importador de datos.

Las clases **GraphVertex** y **GraphEdge** no se mostrarán ya que son relativamente triviales. Sin embargo, resulta interesante resaltar la clase **Node**, la cual contiene información relevante sobre cada uno de los vértices del grafo. Esta clase permite generar instancias de los distintos tipos de nodos, es decir, nodos que permiten la generación y destrucción de los fantasmas de la *demo* planteada.

Listado 6.33: Clase Graph.

```
#include <iostream>
#include <vector>
#include <GraphVertex.h>
#include <GraphEdge.h>

class Graph
{
 public:
  Graph ();
  ~Graph ();

  void addVertex (GraphVertex* pVertex);
  void addEdge (GraphVertex* pOrigin, GraphVertex* pDestination,
        bool undirected = true);

  // Lista de vértices adyacentes a uno dado.
  std::vector<GraphVertex*> adjacents (int index);

  GraphVertex* getVertex (int index);
  std::vector<GraphVertex*> getVertexes () const
    {return _vertexes;}
  std::vector<GraphEdge*> getEdges () const { return _edges; }

 private:
  std::vector<GraphVertex*> _vertexes;
```

```
26   std::vector<GraphEdge*> _edges;
27 };
```

Listado 6.34: Clase Node.

```
1  // SE OMITE PARTE DEL CÓDIGO FUENTE.
2  class Node
3  {
4   public:
5    Node ();
6    Node (const int& index, const string& type,
7      const Ogre::Vector3& pos);
8    ~Node ();
9
10   int getIndex () const { return _index; }
11   string getType () const { return _type; }
12   Ogre::Vector3 getPosition () const { return _position; }
13
14  private:
15   int _index;              // Índice del nodo (id único)
16   string _type;            // Tipo: generador (spawn), sumidero
         (drain)
17   Ogre::Vector3 _position; // Posición del nodo en el espacio 3D
18 };
```

GraphVertex y *Node*

La clase *GraphVertex* mantiene como variable de clase un objeto de tipo *Node*, el cual alberga la información de un nodo en el espacio 3D.

Respecto al diseño de las cámaras virtuales, las clases **Camera** y **Frame** son las utilizadas para encapsular la información y funcionalidad de las mismas. En esencia, una cámara consiste en un identificador, un atributo que determina la tasa de *frames* por segundo a la que se mueve y una secuencia de puntos clave que conforman el camino asociado a la cámara.

La clase *Importer*

API DOM y memoria

El árbol generado por el API DOM a la hora de *parsear* un documento XML puede ser costoso en memoria. En ese caso, se podría plantear otras opciones, como por ejemplo el uso del API SAX.

El punto de interacción entre los datos contenidos en el documento XML y la lógica de dominio previamente discutida está representado por la clase *Importer*. Esta clase proporciona la funcionalidad necesaria para ***parsear* documentos XML** con la estructura planteada anteriormente y *rellenar* las estructuras de datos diseñadas en la anterior sección.

El siguiente listado de código muestra la declaración de la clase *Importer*. Como se puede apreciar, dicha clase hereda de *Ogre::Singleton* para garantizar que solamente existe una instancia de dicha clase, accesible con las funciones miembro *getSingleton()* y *getSingletonPtr()*[13].

Note cómo, además de estas dos funciones miembro, la única función miembro pública es *parseScene()* (línea 9), la cual se puede utilizar para *parsear* un documento XML cuya ruta se especifica en el primer parámetro. El efecto de realizar una llamada a esta función tendrá como resultado el segundo parámetro de la misma, de tipo puntero a objeto de clase *Scene*, con la información obtenida a partir del documento XML (siempre y cuando no se produzca ningún error).

Listado 6.35: Clase Importer.

```
1  #include <OGRE/Ogre.h>
2  #include <xercesc/dom/DOM.hpp>
3  #include <Scene.h>
4
5  class Importer: public Ogre::Singleton<Importer> {
```

[13] Se podría haber desligado la implementación del patrón *Singleton* y Ogre3D.

```
 6  public:
 7   // Única función miembro pública para parsear.
 8   void parseScene (const char* path, Scene *scn);
 9
10   static Importer& getSingleton ();     // Ogre::Singleton.
11   static Importer* getSingletonPtr ();  // Ogre::Singleton.
12
13  private:
14   // Funcionalidad oculta al exterior.
15   // Facilita el parseo de las diversas estructuras
16   // del documento XML.
17   void parseCamera (xercesc::DOMNode* cameraNode, Scene* scn);
18
19   void addPathToCamera (xercesc::DOMNode* pathNode, Camera *cam);
20   void getFramePosition (xercesc::DOMNode* node,
21                          Ogre::Vector3* position);
22   void getFrameRotation (xercesc::DOMNode* node,
23                          Ogre::Vector4* rotation);
24
25   void parseGraph (xercesc::DOMNode* graphNode, Scene* scn);
26   void addVertexToScene (xercesc::DOMNode* vertexNode, Scene* scn);
27   void addEdgeToScene (xercesc::DOMNode* edgeNode, Scene* scn);
28
29   // Función auxiliar para recuperar valores en punto flotante
30   // asociados a una determinada etiqueta (tag).
31   float getValueFromTag (xercesc::DOMNode* node, const XMLCh *tag);
32 };
```

El código del importador se ha estructurado de acuerdo a la definición del propio documento XML, es decir, teniendo en cuenta las etiquetas más relevantes dentro del mismo. Así, dos de las funciones miembro privadas relevantes son, por ejemplo, las funciones *parseCamera()* y *parseGraph()*, usadas para procesar la información de una cámara virtual y del grafo que determina el escenario de la *demo*.

Estructurando código...

Una buena programación estructura es esencial para facilitar el mantenimiento del código y balancear de manera adecuada la complejidad de las distintas funciones que forman parte del mismo.

La biblioteca Xerces-C++ hace uso de tipos de datos específicos para, por ejemplo, manejar cadenas de texto o los distintos nodos del árbol cuando se hace **uso del API DOM**. De hecho, el API utilizada en este importador es DOM. El siguiente listado de código muestra cómo se ha realizado el procesamiento inicial del árbol asociado al documento XML.

Aunque se ha omitido gran parte del código, incluido el tratamiento de errores, el esquema de procesamiento representa muy bien la forma de manejar este tipo de documentos haciendo uso del API DOM. Básicamente, la idea general consiste en obtener una referencia a la raíz del documento (líneas 13-14) y, a partir de ahí, ir recuperando la información contenida en cada uno de los nodos del mismo.

El bucle *for* (líneas 20-21) permite recorrer los nodos hijo del nodo raíz. Si se detecta un nodo con la etiqueta <*camera*>, entonces se llama a la función miembro *parseCamera()*. Si por el contrario se encuentra la etiqueta <*graph*>, entonces se llama a la función *parseGraph()*. En ambos casos, estas funciones irán *poblando* de contenido el puntero a objeto de tipo *Scene* que se pasa como segundo argumento.

Listado 6.36: Clase Importer. Función parseScene

```
 1 // SE OMITE PARTE DEL CÓDIGO FUENTE (bloques try-catch)
 2 void Importer::parseScene (const char* path, Scene *scene) {
 3   // Inicialización.
 4   XMLPlatformUtils::Initialize();
 5
 6   XercesDOMParser* parser = new XercesDOMParser();
 7   parser->parse(path);
 8
 9   DOMDocument* xmlDoc;
10   DOMElement* elementRoot;
11
```

```
12    // Obtener el elemento raíz del documento.
13    xmlDoc = parser->getDocument();
14    elementRoot = xmlDoc->getDocumentElement();
15
16    XMLCh* camera_ch = XMLString::transcode("camera");
17    XMLCh* graph_ch = XMLString::transcode("graph");
18
19    // Procesando los nodos hijos del raíz...
20    for (XMLSize_t i = 0;
21         i < elementRoot->getChildNodes()->getLength(); ++i ) {
22      DOMNode* node = elementRoot->getChildNodes()->item(i);
23
24      if (node->getNodeType() == DOMNode::ELEMENT_NODE) {
25        // Nodo <camera>?
26        if (XMLString::equals(node->getNodeName(), camera_ch))
27      parseCamera(node, scene);
28        else
29      // Nodo <graph>?
30      if (XMLString::equals(node->getNodeName(), graph_ch))
31        parseGraph(node, scene);
32      }
33    }// Fin for
34    // Liberar recursos.
35  }
```

Capítulo 7

Bajo Nivel y Concurrencia

David Vallejo Fernández

Este capítulo realiza un recorrido por los **sistemas de soporte de bajo nivel** necesarios para efectuar diversas tareas críticas para cualquier motor de juegos. Algunos ejemplos representativos de dichos subsistemas están vinculados al arranque y la parada del motor, su configuración y a la gestión de cuestiones de más bajo nivel.

El objetivo principal del presente capítulo consiste en profundizar en dichas tareas y en proporcionar al lector una visión más detallada de los subsistemas básicos sobre los que se apoyan el resto de elementos del motor de juegos.

From the ground up!

Los subsistemas de bajo nivel del motor de juegos resultan esenciales para la adecuada integración de elementos de más alto nivel. Algunos de ellos son simples, pero la funcionalidad que proporcionan es crítica para el correcto funcionamiento de otros subsistemas.

En concreto, en este capítulo se discutirán los siguientes subsistemas:

- Subsistema de **arranque y parada**.
- Subsistema de **gestión de contenedores**.
- Subsistema de **gestión de cadenas**.

El subsistema de gestión relativo a la gestión de contenedores de datos se discutió en el capítulo 5. En este capítulo se llevó a cabo un estudio de la biblioteca STL y se plantearon unos criterios básicos para la utilización de contenedores de datos en el ámbito del desarrollo de videojuegos con C++. Sin embargo, este capítulo dedica una breve sección a algunos aspectos relacionados con los contenedores que no se discutieron anteriormente.

Por otra parte, el subsistema de gestión de memoria se estudiará en el módulo 3, titulado *Técnicas Avanzadas de Desarrollo*, del presente curso de desarrollo de videojuegos. Respecto a esta cuestión particular, se hará especial hincapié en las técnicas y las posibilidades que ofrece C++ en relación a la adecuada gestión de la memoria del sistema.

Así mismo, en este capítulo se aborda la problemática relativa a la **gestión de la concurrencia** mediante un esquema basado en el uso de hilos y de mecanismos típicos de sincronización.

En los últimos años, el desarrollo de los procesadores, tanto de ordenadores personales, consolas de sobremesa e incluso teléfonos móviles, ha estado marcado por un modelo basado en la integración de varios núcleos físicos de ejecución. El objetivo principal de este diseño es la **paralelización** de las tareas a ejecutar por parte del procesador, permitiendo así incrementar el rendimiento global del sistema a través del paralelismo a nivel de ejecución.

Este esquema, unido al concepto de hilo como unidad básica de ejecución de la CPU, ha permitido que aplicaciones tan exigentes como los videojuegos se pueden aprovechar de esta sustancial mejora de rendimiento. Desafortunadamente, no todo son buenas noticias. Este modelo de desarrollo basado en la ejecución concurrente de múltiples hilos de control tiene como consecuencia directa el incremento de la complejidad a la hora de desarrollar dichas aplicaciones.

Además de plantear soluciones que sean paralelizables y escalables respecto al número de unidades de procesamiento, un aspecto crítico a considerar es el **acceso concurrente** a los datos. Por ejemplo, si dos hilos de ejecución distintos comparten un fragmento de datos sobre el que leen y escriben indistintamente, entonces el programador ha de integrar soluciones que garanticen la consistencia de los datos, es decir, soluciones que eviten situaciones en las que un hilo está escribiendo sobre dichos datos y otro está accediendo a los mismos.

Con el objetivo de abordar esta problemática desde un punto de vista práctico en el ámbito del desarrollo de videojuegos, en este capítulo se plantean distintos mecanismos de sincronización de hilos haciendo uso de los **mecanismos nativos ofrecidos en C++11** y, por otra parte, de la **biblioteca de hilos de ZeroC ICE**, un *middleware* de comunicaciones que proporciona una biblioteca de hilos que abstrae al desarrollador de la plataforma y el sistema operativo subyacentes.

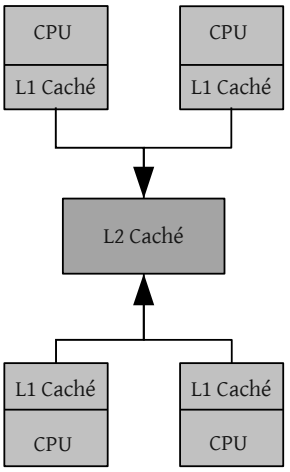


Figura 7.1: Esquema general de una CPU con varios procesadores.

First things first!

El arranque y la parada de subsistemas representa una tarea básica y, al mismo tiempo, esencial para la correcta gestión de los distintos componentes de la arquitectura de un motor de juegos.

7.1. Subsistema de arranque y parada

7.1.1. Aspectos fundamentales

El subsistema de arranque y parada es el responsable de llevar a cabo la **inicialización y configuración** de los distintos subsistemas que forman parte del motor de juegos, así como de realizar la parada de los mismos cuando así sea necesario. Este sistema forma parte de la capa de subsistemas principales que se introdujo brevemente en la sección 1.2.4. La figura 7.2 muestra la interacción del subsistema de arranque y parada con el resto de componentes de la arquitectura general del motor de juegos.

Este subsistema juega un **papel básico** pero fundamental dentro de la arquitectura del motor de juegos, ya que disponer de una entidad software que conozca las interdependencias entre el resto de subsistemas es crucial para efectuar su arranque, configuración y parada de una manera adecuada. Desde un punto de vista general, si un subsistema S tiene una dependencia con respecto a un subsistema T, entonces el subsistema de arranque ha de tener en cuenta dicha dependencia para arrancar primero T y, a continuación, S. Así mismo, la parada de dichos subsistemas se suele realizar generalmente en orden inverso, es decir, primero se pararía S y, posteriormente, T (ver figura 7.3).

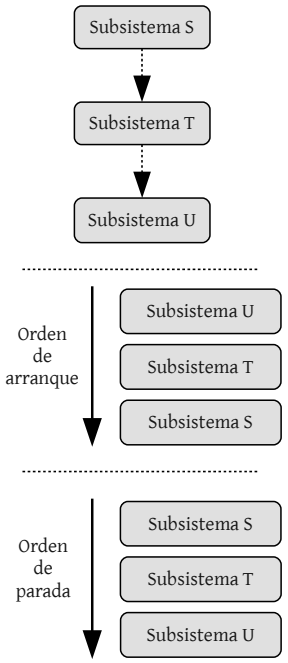


Figura 7.3: Esquema gráfico de orden de arranque y de parada de tres subsistemas dependientes entre sí.

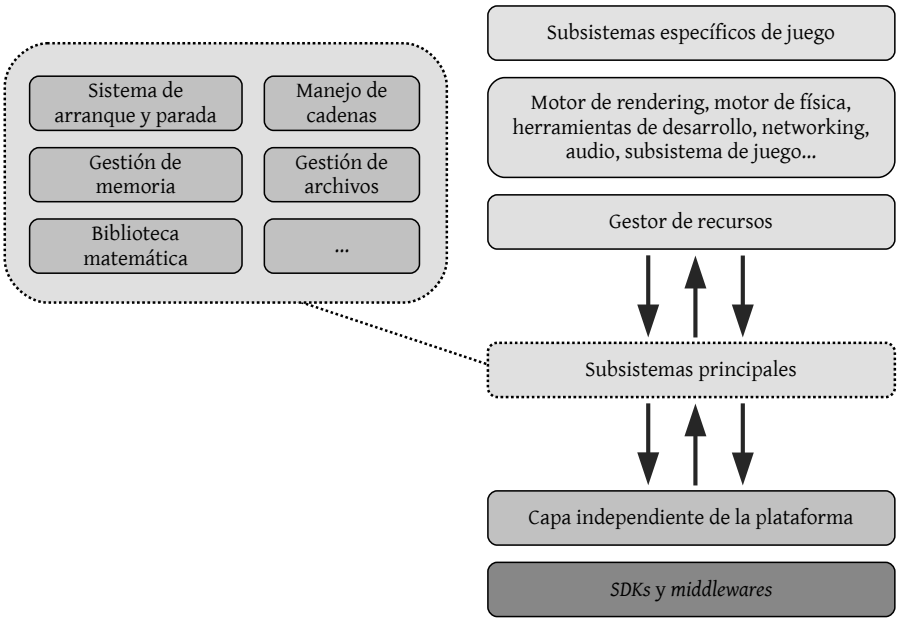


Figura 7.2: Interacción del subsistema de arranque y parada con el resto de subsistemas de la arquitectura general del motor de juegos [42].

En el ámbito del desarrollo de videojuegos, el subsistema de arranque y parada se suele implementar haciendo uso del patrón *singleton*, discutido en la sección 4.2.1, con el objetivo de manejar una única instancia de dicho subsistema que represente el único punto de gestión y evitando la reserva dinámica de memoria. Este planteamiento se extiende a otros subsistemas relevantes que forman parte del motor de juegos. Comúnmente, este tipo de subsistemas también se suelen denominar **gestores o *managers***. La implementación típica de este tipo de elementos se muestra en el siguiente listado de código.

Listado 7.1: Esqueleto básico del subsistema de gestión de memoria.

```
1  class MemoryManager {
2  public:
3    MemoryManager () {
4      // Inicialización gestor de memoria...
5    }
6    ~MemoryManager () {
7      // Parada gestor de memoria...
8    }
9
10   // ...
11 };
12
13 // Instancia única.
14 static MemoryManager gMemoryManager;
```

Variables globales

Recuerde que idealmente las variables globales se deberían limitar en la medida de lo posible con el objetivo de evitar efectos colaterales.

Debido a que C++ es el lenguaje estándar para el desarrollo de videojuegos modernos, una posible alternativa para gestionar de manera correcta tanto el arranque como la parada de subsistemas podría consistir en hacer uso de los mecanismos nativos de C++ de construcción y destrucción de elementos.

Desde un punto de vista general, los **objetos globales y estáticos** se instancian de manera previa a la ejecución de la función *main*. Del mismo modo, dichos objetos se destruyen de manera posterior a la ejecución de dicha función, es decir, inmediatamente después de su retorno. Sin embargo, tanto la construcción como la destrucción de dichos objetos se realizan en un **orden impredecible**.

La consecuencia directa del enfoque nativo proporcionado por C++ para la construcción y destrucción de objetos es que no se puede utilizar como solución directa para arrancar y parar los subsistema del motor de juegos. La razón se debe a que no es posible establecer un orden que considere las posibles interdependencias entre subsistemas.

Listado 7.2: Subsistema de gestión de memoria con acceso a instancia única.

```
1  class MemoryManager {
2  public:
3    static MemoryManager& get () {
4      static MemoryManager gMemoryManager;
5      return gMemoryManager;
6    }
7    MemoryManager () {
8      // Arranque de otros subsistemas dependientes...
9      SubsistemaX::get();
10     SubsistemaY::get();
11
12     // Inicialización gestor de memoria...
13   }
14   ~MemoryManager () {
15     // Parada gestor de memoria...
16   }
17   // ...
18 };
```

Una solución directa a esta problemática consiste en declarar la variable estática dentro de una función, con el objetivo de obtenerla cuando así sea necesario. De este modo, dicha instancia no se instanciará antes de la función *main*, sino que lo hará cuando se efectúe la primera llamada a la función implementada. El anterior listado de código muestra una posible implementación de esta solución.

Una posible variante de este diseño consiste en reservar memoria de manera dinámica para la instancia única del gestor en cuestión, tal y como se muestra en el siguiente listado de código.

Variables no locales

La inicialización de variables estáticas no locales se controla mediante el mecanismo que utilice la implementación para arrancar un programa en C++.

Listado 7.3: Subsistema de gestión de memoria. Reserva dinámica.

```
1  static MemoryManager& get () {
2    static MemoryManager *gMemoryManager = NULL;
3
4    if (gMemoryManager == NULL)
5      gMemoryManager = new MemoryManager();
6
7    assert(gMemoryManager);
8    return *gMemoryManager;
9  }
```

No obstante, aunque existe cierto control sobre el arranque de los subsistemas dependientes de uno en concreto, esta alternativa no proporciona control sobre la parada de los mismos. Así, es posible que C++ destruya uno de los subsistemas dependientes del gestor de memoria de manera previa a la destrucción de dicho subsistema.

Además, este enfoque presenta el inconveniente asociado al momento de la construcción de la instancia global del subsistema de memoria. Evidentemente, la construcción se efectuará a partir de la primera llamada a la función *get*, pero es complicado controlar cuándo se efectuará dicha llamada.

Por otra parte, no es correcto realizar suposiciones sobre la complejidad vinculada a la obtención del *singleton*, ya que los distintos subsistemas tendrán necesidades muy distintas entre sí a la hora de inicializar su estado, considerando las interdependencias con otros subsistemas. En general, este diseño puede ser problemático y es necesario proporcionar un esquema sobre el que se pueda ejercer un mayor control y que simplique los procesos de arranque y parada.

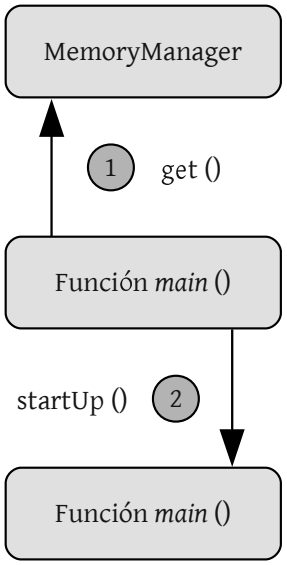


Figura 7.4: Esquema gráfico de obtención de un *singleton* y su posterior utilización.

Interdependencias

El orden del arranque, inicialización y parada de subsistemas dependerán de las relaciones entre los mismos. Por ejemplo, el subsistema de gestión de memoria será típicamente uno de los primeros en arrancarse, debido a que gran parte del resto de subsistemas tienen una dependencia directa respecto al primero.

7.1.2. Esquema típico de arranque y parada

En esta subsección se estudia un enfoque simple [42], aunque ampliamente utilizado, para gestionar tanto el arranque como la parada de los distintos subsistemas que forman la arquitectura de un motor de juegos. La idea principal de este enfoque reside en explicitar el arranque y la parada de dichos subsistemas, que a su vez se gestionan mediante *managers* implementados de acuerdo al patrón *singleton*.

Para ello, tanto el arranque como la parada de un subsistema se implementa mediante funciones explícitas de arranque y parada, típicamente denominadas **startUp** y **shutDown**, respectivamente. En esencia, estas funciones representan al constructor y al destructor de la clase. Sin embargo, la posibilidad de realizar llamadas permite controlar de manera adecuada la inicialización y parada de un subsistema.

El siguiente listado de código muestra la implementación típica de un subsistema de gestión haciendo uso del enfoque planteado en esta subsección. Observe cómo tanto el constructor y el destructor de clase están vacíos, ya que se delega la inicialización del subsistema en las funciones *startUp()* y *shutDown()*.

Listado 7.4: Implementación de funciones startUp() y shutDown().

```
1  class MemoryManager {
2  public:
3    MemoryManager () {}
4    ~MemoryManager () {}
5
6    void startUp () {
7      // Inicialización gestor de memoria...
8    }
9    void shutDown () {
10     // Parada gestor de memoria...
11   }
12 };
13
14 class RenderManager { /* IDEM */ };
15 class TextureManager { /* IDEM */ };
16 class AnimationManager { /* IDEM */ };
17 // ...
18
19 MemoryManager gMemoryManager;
20 RenderManager gRenderManager;
21 TextureManager gTextureManager;
22 AnimationManager gAnimationManager;
23 // ...
```

Recuerde que la simplicidad suele ser deseable en la mayoría de los casos, aunque el rendimiento o el propio diseño de un programa se vean degradados. Este enfoque facilita el mantenimiento del código. El subsistema de arranque y parada es un caso típico en el ámbito de desarrollo de videojuegos.

Mediante este enfoque, la inicialización y parada de subsistemas es trivial y permite un mayor control sobre dichas tareas. En el siguiente listado de código se muestran de manera explícita las dependencias entre algunos de los subsistemas típicos de la arquitectura de motor, como por ejemplo los subsistemas de gestión de memoria, manejo de texturas, renderizado o animación.

Listado 7.5: Arranque y parada de subsistemas típicos.

```
1  int main () {
2    // Arranque de subsistemas en orden.
3    gMemoryManager.startUp();
4    gTextureManager.startUp();
5    gRenderManager.startUp();
6    gAnimationManager.startUp();
7    // ...
8
9    // Bucle principal.
10   gSimulationManager.run();
11
12   // Parada de subsistemas en orden.
13   // ...
14   gAnimationManager.startUp();
15   gRenderManager.startUp();
16   gTextureManager.startUp();
17   gMemoryManager.startUp();
18
19   return 0;
20 }
```

7.1.3. Caso de estudio. Ogre 3D

Aunque Ogre 3D es en realidad un motor de renderizado en lugar de un completo motor de juegos, dicho entorno proporciona una gran cantidad de subsistemas relevantes para facilitar el desarrollo de videojuegos. Entre ellos, también existe un subsistema de arranque y parada que hace gala de una gran simplicidad y sencillez.

Básicamente, el arranque y la parada en Ogre se realiza a través del uso de la clase **Ogre::Root**, la cual implementa el patrón *singleton* con el objetivo de asegurar una única instancia de dicha clase, la cual actúa como punto central de gestión.

Para ello, la clase *Root* almacena punteros, como variables miembro privadas, a todos los subsistemas soportados por Ogre con el objetivo de gestionar su creación y su destrucción. El siguiente listado de código muestra algunos aspectos relevantes de la declaración de esta clase.

Figura 7.5: Aunque Ogre no es un motor de juegos propiamente dicho, su complejidad hace necesaria la integración de una gran cantidad de subsistemas. Dichos subsistemas han de permitir la adecuada gestión de su arranque y su parada.

Ogre::Singleton

La clase *Ogre::Singleton* está basada en el uso de plantillas para poder instanciarla para tipos particulares. Dicha clase proporciona las típicas funciones *getSingleton()* y *getSingletonPtr* para acceder a la referencia y al puntero, respectivamente, de la única instancia de clase creada.

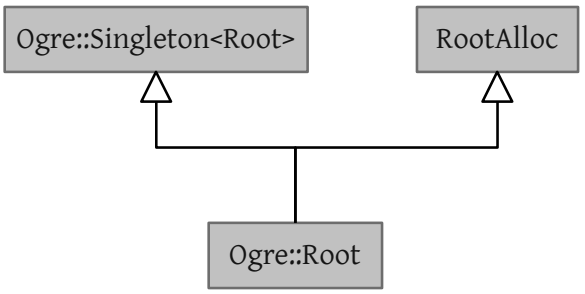


Figura 7.6: Clase *Ogre::Root* y su relación con las clases *Ogre::Singleton* y *RootAlloc*.

Como se puede apreciar, la clase *Root* hereda de la clase *Ogre::Singleton*, que a su vez hace uso de plantillas y que, en este caso, se utiliza para especificar el *singleton* asociado a la clase *Root*. Así mismo, dicha clase hereda de *RootAlloc*, que se utiliza como superclase para todos los objetos que deseen utilizar un asignador de memoria personalizado.

Listado 7.6: Clase Ogre::Root.

```
1  class _OgreExport Root : public Singleton<Root>, public RootAlloc
2  {
3      protected:
4          // Más declaraciones...
5          // Gestores (implementan el patrón Singleton).
6          LogManager* mLogManager;
7          ControllerManager* mControllerManager;
8          SceneManagerEnumerator* mSceneManagerEnum;
9          DynLibManager* mDynLibManager;
10         ArchiveManager* mArchiveManager;
11         MaterialManager* mMaterialManager;
12         MeshManager* mMeshManager;
13         ParticleSystemManager* mParticleManager;
14         SkeletonManager* mSkeletonManager;
15         OverlayElementFactory* mPanelFactory;
16         OverlayElementFactory* mBorderPanelFactory;
17         OverlayElementFactory* mTextAreaFactory;
18         OverlayManager* mOverlayManager;
19         FontManager* mFontManager;
20         ArchiveFactory *mZipArchiveFactory;
21         ArchiveFactory *mFileSystemArchiveFactory;
22         ResourceGroupManager* mResourceGroupManager;
23         ResourceBackgroundQueue* mResourceBackgroundQueue;
24         ShadowTextureManager* mShadowTextureManager;
25         RenderSystemCapabilitiesManager*
               mRenderSystemCapabilitiesManager;
26         ScriptCompilerManager *mCompilerManager;
27     // Más declaraciones...
28 };
```

El objeto *Root* representa el **punto de entrada** de Ogre y ha de ser el primer objeto instanciado en una aplicación y el último en ser destruido. Típicamente, la clase principal de los ejemplos básicos desarrollados tendrán como variable miembro una instancia de la clase *Root*, con el objetivo de facilitar la administración del juego en cuestión.

Desde el punto de vista del renderizado, el objeto de tipo *Root* proporciona la función *startRendering*. Cuando se realiza una llamada a dicha función, la aplicación entrará en un bucle de renderizado continuo que finalizará cuando todas las ventanas gráficas se hayan cerrado o cuando todos los objetos del tipo *FrameListener* finalicen su ejecución (ver módulo 2, *Programación Gráfica*).

La implementación del constructor de la clase *Ogre::Root* tiene como objetivo principal instanciar y arrancar los distintos subsistemas previamente declarados en el anterior listado de código. A continuación se muestran algunos aspectos relevantes de la implementación de dicho constructor.

Start your engine!

Aunque el arranque y la parada de los motores de juegos actuales suelen estar centralizados mediante la gestión de algún elemento central, como por ejemplo la clase *Ogre::Root*, es responsabilidad del desarrollador conocer los elementos accesibles desde dicha entidad de control central.

Listado 7.7: Clase Ogre::Root. Constructor.

```
1  Root::Root(const String& pluginFileName, const String&
       configFileName,
2          const String& logFileName)
3  // Inicializaciones...
4  {
5    // Comprobación del singleton en clase padre.
6    // Inicialización.
7    // src ...
8
9    // Creación del log manager y archivo de log por defecto.
10   if(LogManager::getSingletonPtr() == 0) {
11       mLogManager = OGRE_NEW LogManager();
12       mLogManager->createLog(logFileName, true, true);
13     }
14
15   // Gestor de biblioteca dinámica.
16   mDynLibManager = OGRE_NEW DynLibManager();
17   mArchiveManager = OGRE_NEW ArchiveManager();
18
19   // ResourceGroupManager.
20   mResourceGroupManager = OGRE_NEW ResourceGroupManager();
21
22   // src...
23
24   // Material manager.
25   mMaterialManager = OGRE_NEW MaterialManager();
26   // Mesh manager.
27   mMeshManager = OGRE_NEW MeshManager();
28   // Skeleton manager.
29   mSkeletonManager = OGRE_NEW SkeletonManager();
30   // Particle system manager.
31   mParticleManager = OGRE_NEW ParticleSystemManager();
32   // src...
33 }
```

Figura 7.7: *Quake III Arena* es un juego multijugador en primera persona desarrollado por IdSoftware y lanzado el 2 de Diciembre de 1999. El modo de juego online fue uno de los más populares en su época.

7.1.4. Caso de estudio. Quake III

El código fuente de Quake III[1] fue liberado bajo licencia GPLv2 el día 20 de Agosto de 2005. Desde entonces, la comunidad *amateur* de desarrollo ha realizado modificaciones y mejoras e incluso el propio motor se ha reutilizado para el desarrollo de otros juegos. El diseño de los motores de Quake es un muy **buen ejemplo** de arquitectura bien estructurada y modularizada, hecho que posibilita el estudio de su código y la adquisición de experiencia por el desarrollador de videojuegos.

En esta sección se estudiará desde un punto de vista general el sistema de arranque y de parada de Quake III. Al igual que ocurre en el caso de Ogre, el esquema planteado consiste en hacer uso de funciones específicas para el arranque, inicialización y parada de las distintas entidades o subsistemas involucrados.

El siguiente listado de código muestra el punto de entrada de *Quake III* para sistemas *Windows*™. Como se puede apreciar, la función principal consiste en una serie de casos que determinan el flujo de control del juego. Es importante destacar los casos **GAME_INIT** y **GAME_SHUTDOWN** que, como su nombre indica, están vinculados al arranque y la parada del juego mediante las funciones **G_InitGame()** y **G_ShutdownGame()**, respectivamente.

20 de Agosto

La fecha de liberación del código de Quake III no es casual. En realidad, coincide con el cumpleaños de John Carmack, nacido el 20 de Agosto de 1970. Actualmente, John Carmack es una de las figuras más reconocidas dentro del ámbito del desarrollo de videojuegos.

Listado 7.8: Quake 3. Función vmMain().

```
1  int vmMain( int command, int arg0, int arg1, ..., int arg11  ) {
2    switch ( command ) {
3      // Se omiten algunos casos.
4    case GAME_INIT:
5      G_InitGame( arg0, arg1, arg2 );
6      return 0;
7    case GAME_SHUTDOWN:
8      G_ShutdownGame( arg0 );
9      return 0;
10   case GAME_CLIENT_CONNECT:
11     return (int)ClientConnect( arg0, arg1, arg2 );
12   case GAME_CLIENT_DISCONNECT:
13     ClientDisconnect( arg0 );
14     return 0;
15   case GAME_CLIENT_BEGIN:
16     ClientBegin( arg0 );
17     return 0;
18   case GAME_RUN_FRAME:
19     G_RunFrame( arg0 );
20     return 0;
21   case BOTAI_START_FRAME:
22     return BotAIStartFrame( arg0 );
23   }
24
25   return -1;
26 }
```

Recuerde que Quake III está desarrollado utilizando el lenguaje C. El código fuente está bien diseñado y estructurado. Para tener una visión global de su estructura se recomienda visualizar el archivo *g_local.h*, en el que se explicita el fichero de las funciones más relevantes del código.

[1] https://github.com/id-Software/Quake-III-Arena

A continuación se muestran los aspectos más destacados de la función de inicialización de *Quake III*. Como se puede apreciar, la función *G_InitGame()* se encarga de inicializar los aspectos más relevantes a la hora de arrancar el juego. Por ejemplo, se inicializan punteros relevantes para manejar elementos del juego en *G_InitMemory()*, se resetean valores si la sesión de juego ha cambiado en *G_InitWorldSession()*, se inicializan las entidades del juego, etcétera.

Listado 7.9: Quake 3. Función G_InitGame.

```
1  void G_InitGame( int levelTime, int randomSeed, int restart ) {
2    // Se omite parte del código fuente...
3
4    G_InitMemory();
5
6    // ...
7
8    G_InitWorldSession();
9
10   // Inicialización de entidades del juego.
11   memset( g_entities, 0, MAX_GENTITIES * sizeof(g_entities[0]) );
12   level.gentities = g_entities;
13
14   // Inicialización de los clientes.
15   level.maxclients = g_maxclients.integer;
16   memset( g_clients, 0, MAX_CLIENTS * sizeof(g_clients[0]) );
17   level.clients = g_clients;
18
19   // Establecimiento de campos cuando entra un cliente.
20   for ( i=0 ; i<level.maxclients ; i++ ) {
21     g_entities[i].client = level.clients + i;
22   }
23
24   // Reservar puntos para jugadores eliminados.
25   InitBodyQue();
26   // Inicialización general.
27   G_FindTeams();
28
29   // ...
30 }
```

La filosofía seguida para parar de manera adecuada el juego es análoga al arranca e inicialización, es decir, se basa en centralizar dicha funcionalidad en sendas funciones. En este caso concreto, la función *G_ShutdownGame()* es la responsable de llevar a cabo dicha parada.

Listado 7.10: Quake 3. Función G_ShutdownGame.

```
1  void G_ShutdownGame( int restart ) {
2    G_Printf ("==== ShutdownGame ====\n");
3    if ( level.logFile ) {
4      G_LogPrintf("ShutdownGame:\n" );
5      trap_FS_FCloseFile( level.logFile );
6    }
7
8    // Escritura de los datos de sesión de los clientes,
9    // para su posterior recuperación.
10   G_WriteSessionData();
11
12   if ( trap_Cvar_VariableIntegerValue( "bot_enable" ) ) {
```

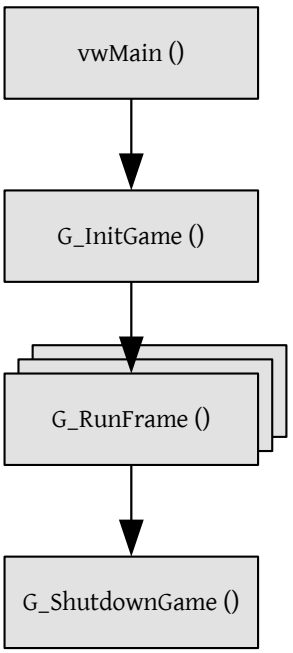


Figura 7.8: Visión abstracta del flujo general de ejecución de *Quake III Arena* (no se considera la conexión y desconexión de clientes).

```
13      BotAIShutdown( restart );
14    }
15  }
```

Como se puede apreciar en la función de parada, básicamente se escribe información en el fichero de *log* y se almacenan los datos de las sesiones de los clientes para, posteriormente, permitir su recuperación. Note cómo en las líneas 14-15 se delega la parada de los bots, es decir, de los NPCs en la función *BotAIShutdown()*. De nuevo, el esquema planteado se basa en delegar la parada en distintas funciones en función de la entidad afectada.

Programación estructura

El código de *Quake* es un buen ejemplo de programación estructurada que gira en torno a una adecuada definición de funciones y a un equilibrio respecto a la complejidad de las mismas.

A su vez, la función *BotAIShutdown()* delega en otra función que es la encargada, finalmente, de liberar la memoria y los recursos previamente reservados para el caso particular de los jugadores que participan en el juego, utilizando para ello la función *BotAIShutdownClient()*.

Listado 7.11: Quake 3. Funciones BotAIShutdown y BotAIShutdownClient.

```
1  int BotAIShutdown( int restart ) {
2    // Si el juego se resetea para torneo...
3    if ( restart ) {
4      // Parar todos los bots en botlib.
5      for (i = 0; i < MAX_CLIENTS; i++)
6        if (botstates[i] && botstates[i]->inuse)
7      BotAIShutdownClient(botstates[i]->client, restart);
8    }
9    // ...
10 }
11
12 int BotAIShutdownClient(int client, qboolean restart) {
13   // Se omite parte del código.
14   // Liberar armas...
15   trap_BotFreeWeaponState(bs->ws);
16   // Liberar el bot...
17   trap_BotFreeCharacter(bs->character);
18   // ..
19   // Liberar el estado del bot...
20   memset(bs, 0, sizeof(bot_state_t));
21   // Un bot menos...
22   numbots--;
23   // ...
24   // Todo OK.
25   return qtrue;
26 }
```

7.2. Contenedores

Como ya se introdujo en el capítulo 5, los contenedores son simplemente objetos que contienen otros objetos. En el mundo del desarrollo de videojuegos, y en el de las aplicaciones software en general, los contenedores se utilizan extensivamente para almacenar las estructuras de datos que conforman la base del diseño que soluciona un determinado problema.

Algunos de los contenedores más conocidos ya se comentaron en las secciones 5.3, 5.4 y 5.5. En dichas secciones se hizo un especial hincapié en relación a la utilización de estos contenedores, atendiendo a sus principales caracteristicas, como la definición subyacente en la biblioteca STL.

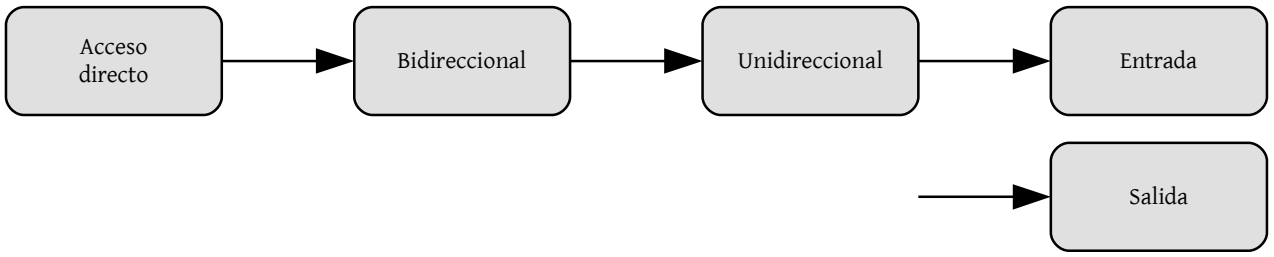


Figura 7.9: Esquema gráfica de las relaciones entre las distintas categorías de iteradores en STL. Cada categoría implementa la funcionalidad de todas las categorías que se encuentran a su derecha.

En esta sección se introducirán algunos aspectos fundamentales que no se discutieron anteriormente y se mencionarán algunas bibliotecas útiles para el uso de algunas estructuras de datos que son esenciales en el desarrollo de videojuegos, como por ejemplo los **grafos**. Estos aspectos se estudiarán con mayor profundidad en el módulo 3, *Técnicas Avanzadas de Desarrollo*.

7.2.1. Iteradores

Desde un punto de vista abstracto, un iterador se puede definir como una clase que permite acceder de manera eficiente a los elementos de un contenedor específico. Según [94], un iterador es una abstracción pura, es decir, cualquier elemento que se comporte como un iterador se define como un iterador. En otras palabras, un iterador es una abstracción del concepto de puntero a un elemento de una secuencia. Los principales elementos clave de un iterador son los siguientes:

Patrón *Iterator*

Normalmente, los iteradores se implementan haciendo uso del patrón que lleva su mismo nombre, tal y como se discutió en la sección 4.4.3.

- El elemento al que apunta (desreferenciado mediante los operadores ***** y **->**).
- La posibilidad de apuntar al siguiente elemento (incrementándolo mediante el operador **++**).
- La igualdad (representada por el operador **==**).

De este modo, un elemento primitivo **int*** se puede definir como un iterador sobre un array de enteros, es decir, sobre **int[]**. Así mismo, **std::list<std::string>::iterator** es un iterador sobre la clase **list**.

Rasgos de iterador

En STL, los tipos relacionados con un iterador se describen a partir de una serie de declaraciones en la plantilla de clase *iterator_traits*. Por ejemplo, es posible acceder al tipo de elemento manejado o el tipo de las operaciones soportadas.

Un iterador representa la abstracción de un puntero dentro de un array, de manera que no existe el concepto de iterador *nulo*. Como ya se introdujo anteriormente, la condición para determinar si un iterador apunta o no a un determinado elemento se evalúa mediante una comparación al elemento final de una secuencia (*end*).

Las principales **ventajas** de utilizar un iterador respecto a intentar el acceso sobre un elemento de un contenedor son las siguientes [42]:

- El acceso directo rompe la encapsulación de la clase contenedora. Por el contrario, el iterador suele ser *amigo* de dicha clase y puede iterar de manera eficiente sin exponer detalles de implementación. De hecho, la mayoría de los contenedores ocultan los detalles internos de implementación y no se puede iterar sobre ellos sin un iterador.
- El iterador simplifica el proceso de iteración. La mayoría de iteradores actúan como índices o punteros, permitiendo el uso de estructuras de bucle para recorrer un contenedor mediante operadores de incremento y comparación.

Categorías de iteradores y operaciones

Categoría	Salida	Entrada	Unidireccional	Bidireccional	Acceso directo
Lectura		=*p	=*p	=*p	=*p
Acceso		->	->	->	->[]
Escritura	*p=		*p=	*p=	*p=
Iteración	++	++	++	++ –	++ – + - += -=
Comparación		== !=	== !=	== !=	== != <><= >=

Cuadro 7.1: Resumen de las principales categorías de iteradores en STL junto con sus operaciones [94].

Dentro de la biblioteca STL existen **distintos tipos de contenedores** y no todos ellos mantienen el mismo juego de operaciones. Los iteradores se clasifican en cinco categorías diferentes dependiendo de las operaciones que proporcionan de manera eficiente, es decir, en tiempo constante ($O(1)$). La tabla 7.1 resume las distintas categorías de iteradores junto con sus operaciones[2].

Recuerde que tanto la lectura como la escritura se realizan mediante el iterador desreferenciado por el operador *****. Además, de manera independiente a su categoría, un iterador puede soportar acceso *const* respecto al objeto al que apunta. No es posible escribir sobre un determinado elemento utilizando para ello un operador *const*, sea cual sea la categoría del iterador.

No olvide implementar el constructor de copia y definir el operador de asignación para el tipo que se utilizará en un contenedor y que será manipulado mediante iteradores, ya que las lecturas y escrituras copian objetos.

Los iteradores también se pueden aplicar sobre **flujos** de E/S gracias a que la biblioteca estándar proporciona cuatro tipos de iteradores enmarcados en el esquema general de contenedores y algoritmos:

- *ostream_iterator*, para escribir en un *ostream*.
- *istream_iterator*, para leer de un *istream*.
- *ostreambuf_iterator*, para escribir en un *buffer* de flujo.
- *istreambuf_iterator*, para leer de un *buffer* de flujo.

El siguiente listado de código muestra un ejemplo en el que se utiliza un iterador para escribir sobre la salida estándar. Como se puede apreciar, el operador ++ sirve para desplazar el iterador de manera que sea posible llevar a cabo dos asignaciones consecutivas sobre el propio flujo. De no ser así, el código no sería portable.

Uso comercial de STL

Aunque STL se ha utilizado en el desarrollo de videojuegos comerciales, actualmente es más común que el motor de juegos haga uso de una biblioteca propia para la gestión de contenedores básicos. Sin embargo, es bastante común encontrar esquemas que siguen la filosofía de STL, aunque optimizados en aspectos críticos como por ejemplo la gestión y asignación de memoria.

7.2.2. Más allá de STL

Como ya se discutió anteriormente en el capítulo 5, gran parte de los motores de juego tienen sus propias implementaciones de los contenedores más utilizados para el manejo de sus estructuras de datos. Este planteamiento está muy extendido en el ámbito de las consolas de sobremesa, las consolas portátiles, los teléfonos móviles y las PDA (Personal Digital Assistant)s. Los principales motivos son los siguientes:

[2] `http://www.cplusplus.com/reference/std/iterator/`

Listado 7.12: Ejemplo de uso de iteradores para llevar a cabo operaciones de E/S.

```
1  #include <iostream>
2  #include <iterator>
3
4  using namespace std;
5
6  int main () {
7    // Escritura de enteros en cout.
8    ostream_iterator<int> fs(cout);
9
10   *fs = 7;  // Escribe 7 (usa cout).
11   ++fs;     // Preparado para siguiente salida.
12   *fs = 6;  // Escribe 6.
13
14   return 0;
15 }
```

- **Control total** sobre los contenedores y estructuras de datos desarrolladas, especialmente sobre el mecanismo de asignación de memoria, aunque sin olvidar aspectos como los propios algoritmos. Aunque STL permite crear asignadores de memoria personalizados (*custom allocators*), en ocasiones los propios patrones de los contenedores de STL pueden ser insuficientes.
- **Optimizaciones**, considerando el propio *hardware* sobre el que se ejecutará el motor. STL es un estándar independiente de la plataforma y el sistema operativo, por lo que no es posible aplicar optimizaciones de manera directa sobre la propia biblioteca.
- **Personalización** debido a la necesidad de incluir funcionalidad sobre un contenedor que no esté inicialmente considerada en otras bibliotecas como STL. Por ejemplo, en un determinado problema puede ser necesario obtener los n elementos más adecuados para satisfacer una necesidad.
- **Independencia funcional**, ya que es posible depurar y arreglar cualquier problema sin necesidad de esperar a que sea solucionado por terceras partes.

De manera independiente al hecho de considerar la opción de usar STL, existen otras bibliotecas relacionadas que pueden resultar útiles para el desarrollador de videojuegos. Una de ellas es **STLPort**, una implementación de STL específicamente ideada para ser portable a un amplio rango de compiladores y plataformas. Esta implementación proporciona una funcionalidad más rica que la que se puede encontrar en otras implementaciones de STL.

Estandarizando *Boost*

Gran parte de las bibliotecas del proyecto *Boost* están en proceso de estandarización con el objetivo de incluirse en el propio estándar de C++.

La otra opción que se discutirá brevemente en esta sección es el proyecto **Boost**, cuyo principal objetivo es extender la funcionalidad de STL. Las principales características de *Boost* se resumen a continuación:

- Inclusión de funcionalidad no disponible en STL.
- En ocasiones, *Boost* proporciona algunas alternativas de diseño e implementación respecto a STL.
- Gestión de aspectos complejos, como por ejemplos los *smart pointers*.
- Documentación de gran calidad que también incluye discusiones sobre las decisiones de diseño tomadas.

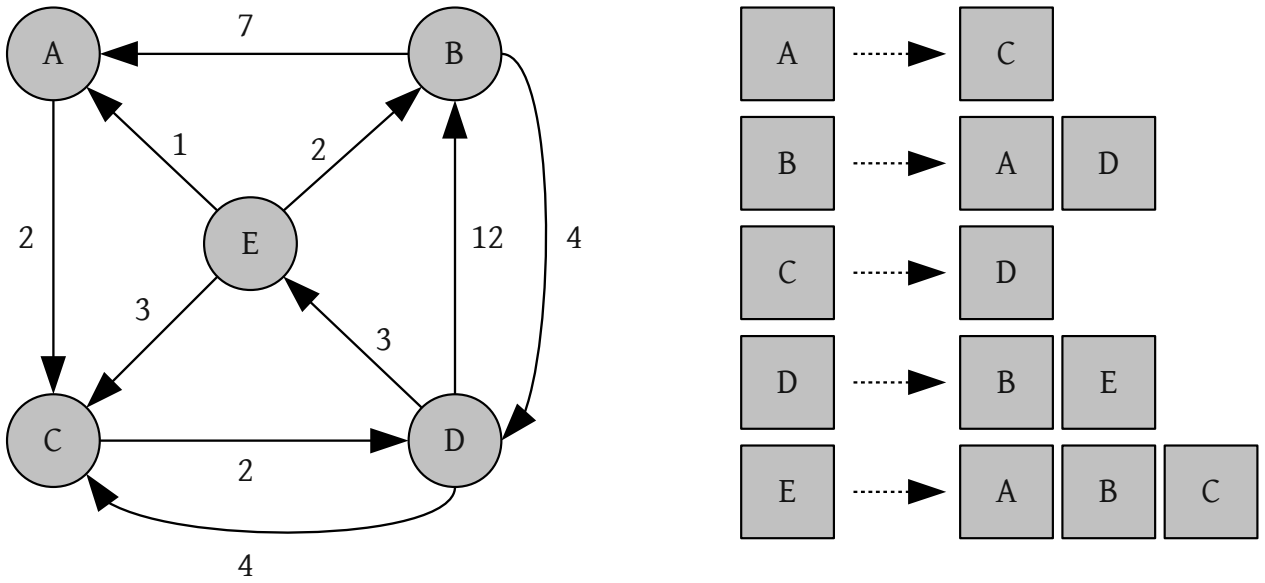


Figura 7.10: Representación gráfica del grafo de ejemplo utilizado para el cálculo de caminos mínimos. La parte derecha muestra la implementación mediante listas de adyacencia planteada en la biblioteca *Boost*.

Para llevar a cabo la instalación de *STLPort*[3] y *Boost*[4] (incluída la biblioteca para manejo de grafos) en sistemas operativos Debian y derivados es necesarios ejecutar los siguientes comandos:

```
$ sudo apt-get update
$ sudo apt-get install libstlport5.2-dev
$ sudo apt-get install libboost-dev
  $ sudo apt-get install libboost-graph-dev
```

La figura 7.10 muestra un grafo dirigido que servirá como base para la discusión del siguiente fragmento de código, en el cual se hace uso de la biblioteca *Boost* para llevar a cabo el cálculo de los caminos mínimos desde un vértice al resto mediante el algoritmo de *Dijkstra*.

Las bibliotecas de *Boost* se distribuyen bajo la *Boost Software Licence*, la cual permite el uso comercial y no-comercial.

Edsger W. Dijkstra

Una de las personas más relevantes en el ámbito de la computación es *Dijkstra*. Entre sus aportaciones, destacan la solución al camino más corto, la notación polaca inversa, el algoritmo del banquero, la definición de semáforo como mecanismo de sincronización e incluso aspectos de computación distribuida, entre otras muchas contribuciones.

Recuerde que un **grafo** es un conjunto no vacío de nodos o vértices y un conjunto de pares de vértices o aristas que unen dichos nodos. Un grafo puede ser dirigido o no dirigido, en función de si las aristas son unidireccionales o bidireccionales, y es posible asignar pesos a las aristas, conformando así un grafo valorado.

[3] http://www.stlport.org
[4] http://www.boost.org

A continuación se muestra un listado de código (adaptado a partir de uno de los ejemplos de la biblioteca[5]) que permite la definición básica de un grafo dirigido y valorado. El código también incluye cómo hacer uso de funciones relevantes asociadas los grafos, como el cálculo de los cáminos mínimos de un determinado nodo al resto.

Como se puede apreciar en el siguiente código, es posible diferenciar cuatro bloques básicos. Los tres primeros se suelen repetir a la hora de manejar estructuras mediante la biblioteca *Boost Graph*. En primer lugar, se lleva a cabo la definición de los tipos de datos que se utilizarán, destacando el propio grafo (líneas 15–16) y los descriptores para manejar tanto los vértices (línea 17) y las aristas (línea 18). A continuación se especifican los elementos concretos del grafo, es decir, las etiquetas textuales asociadas a los vértices, las aristas o arcos y los pesos de las mismas. Finalmente, ya es posible instanciar el grafo (línea 32), el descriptor usado para especificar el cálculo de caminos mínimos desde A (línea 38) y la llamada a la función para obtener dichos caminos (41).

Listado 7.13: Ejemplo de uso de la biblioteca Boost (grafos).

```
1  #include <boost/config.hpp>
2  #include <iostream>
3
4  #include <boost/graph/graph_traits.hpp>
5  #include <boost/graph/adjacency_list.hpp>
6  #include <boost/graph/dijkstra_shortest_paths.hpp>
7
8  using namespace boost;
9
10 int
11 main(int, char *[])
12 {
13   // Definición de estructuras de datos...
14   typedef adjacency_list <listS, vecS, directedS,
15     no_property, property <edge_weight_t, int> > graph_t;
16   typedef graph_traits <graph_t>::vertex_descriptor
         vertex_descriptor;
17   typedef graph_traits <graph_t>::edge_descriptor edge_descriptor;
18   typedef std::pair<int, int> Edge;
19
20   // Parámetros básicos del grafo.
21   const int num_nodes = 5;
22   enum nodes {A, B, C, D, E};
23   char name[] = "ABCDE";
24   Edge edge_array[] = {
25     Edge(A, C), Edge(B, A), Edge(B, D), Edge(C, D), Edge(D, B),
26     Edge(D, C), Edge(D, E), Edge(E, A), Edge(E, B), Edge(E, C)
27   };
28   int weights[] = {2, 7, 4, 2, 12, 4, 3, 1, 2, 3};
29   int num_arcs = sizeof(edge_array) / sizeof(Edge);
30
31   // Instanciación del grafo.
32   graph_t g(edge_array, edge_array + num_arcs, weights, num_nodes);
33   property_map<graph_t, edge_weight_t>::type weightmap =
34     get(edge_weight, g);
35
36   std::vector<vertex_descriptor> p(num_vertices(g));
37   std::vector<int> d(num_vertices(g));
38   vertex_descriptor s = vertex(A, g); // CAMINOS DESDE A.
39
40   // Algoritmo de Dijkstra.
41   dijkstra_shortest_paths(g, s,
42               predecessor_map(&p[0]).distance_map(&d[0]));
43
44   std::cout << "Distancias y nodos padre:" << std::endl;
```

[5] http://www.boost.org/doc/libs/1_55_0/libs/graph/example/dijkstra-example.cpp

```
45   graph_traits <graph_t>::vertex_iterator vi, vend;
46   for (boost::tie(vi, vend) = vertices(g); vi != vend; ++vi) {
47     std::cout << "Distancia(" << name[*vi] << ") = "
48           << d[*vi] << ", ";
49     std::cout << "Padre(" << name[*vi] << ") = " << name[p[*vi]]
50           << std:: endl;
51   }
52   std::cout << std::endl;
53
54   return EXIT_SUCCESS;
55 }
```


El último bloque de código (líneas 44-52) permite obtener la información relevante a partir del cálculo de los caminos mínimos desde A, es decir, la distancia que existe desde el propio nodo A hasta cualquier otro y el nodo a partir del cual se accede al destino final (partiendo de A).

Para generar un ejecutable, simplemente es necesario enlazar con la biblioteca *boost_graph*:

```
$ g++ dijkstra.cpp -o dijkstra -lboost_graph
$ ./dijkstra
```

Al ejecutar, la salida del programa mostrará todos los caminos mínimos desde el vértice especificado; en este caso, desde el vértice A.

```
Distancias y nodos padre:

Distancia(A) = 0, Padre(A) = A
Distancia(B) = 9, Padre(B) = E
Distancia(C) = 2, Padre(C) = A
Distancia(D) = 4, Padre(D) = C
Distancia(E) = 7, Padre(E) = D
```

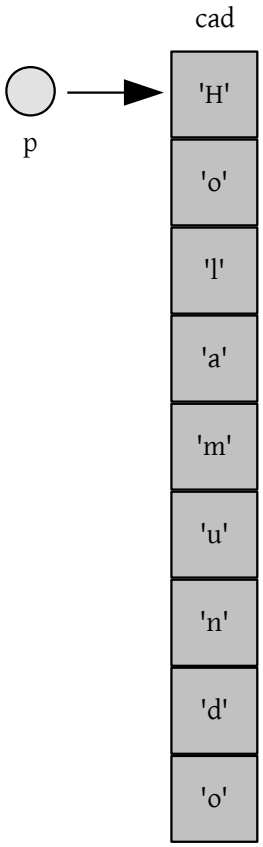


Figura 7.11: Esquema gráfico de la implementación de una cadena mediante un array de caracteres.

7.3. Subsistema de gestión de cadenas

Al igual que ocurre con otros elementos transversales a la arquitectura de un motor de juegos, como por ejemplos los contenedores de datos (ver capítulo 5), las **cadenas de texto** se utilizan extensivamente en cualquier proyecto software vinculado al desarrollo de videojuegos. Aunque en un principio pueda parecer que la utilización y la gestión de cadenas de texto es una cuestión trivial, en realidad existe una gran variedad de técnicas y restricciones vinculadas a este tipo de datos. Su consideración puede ser muy relevante a la hora de mejorar el rendimiento de la aplicación.

7.3.1. Cuestiones específicas

Uno de los aspectos críticos a la hora de tratar con cadenas de texto está vinculado con el **almacenamiento y gestión** de las mismas. Si se considera el hecho de utilizar C/C++ para desarrollar un videojuego, entonces es importante no olvidar que en estos lenguajes de programación las cadenas de texto no son un tipo de datos atómica, ya que se implementan mediante un **array de caracteres**.

```
typedef basic_string<char> string;
```

La consecuencia directa de esta implementación es que el desarrollador ha de considerar cómo gestionar la reserva de memoria para dar soporte al uso de cadenas:

- De manera **estática**, es decir, reservando una cantidad fija e inicial de memoria para el array de caracteres.
- De manera **dinámica**, es decir, asignando en tiempo de ejecución la cantidad de memoria que se necesite para cubrir una necesidad.

Típicamente, la opción más utilizada por los programadores de C++ consiste en hacer uso de una clase *string*. En particular, la clase *string* proporcionada por la biblioteca estándar de C++. Sin embargo, en el contexto del desarrollo de videojuegos es posible que el desarrollador haya decidido no hacer uso de STL, por lo que sería necesario llevar a cabo una implementación nativa.

Internationalization

La localización, en el ámbito de gestión de múltiples lenguajes, se suele conocer como *internationalization*, o I18N, y es esencial para garantizar la inexistencia de problemas al tratar con diferentes idiomas.

Otro aspecto fundamental relativo a las cadenas es la **localización**, es decir, el proceso de adaptar el software desarrollado a múltiples lenguajes. Básicamente, cualquier representación textual del juego, normalmente en inglés en un principio, ha de traducirse a los idiomas soportados por el juego. Esta cuestión plantea una problemática variada, como por ejemplo la necesidad de tener en cuenta diversos alfabetos con caracteres específicos (por ejemplo chino o japonés), la posibilidad de que el texto tenga una orientación distinta a la occidental o aspectos más específicos, como por ejemplo que la traducción de un término tenga una longitud significativamente más larga, en términos de longitud de caracteres, que su forma original.

Desde un punto de vista más interno al propio desarrollo, también hay que considerar que las cadenas se utilizan para identificar las distintas entidades del juego. Por ejemplo, es bastante común adoptar un convenio para nombrar a los posibles enemigos del personaje principal en el juego. Un ejemplo podría ser *flying-enemy-robot-02*.

El tratamiento de cadenas en tiempo de ejecución es costoso. Por lo tanto, evaluar su impacto es esencial para obtener un buen rendimiento.

Finalmente, el impacto de las **operaciones típicas** de cadenas es muy importante con el objetivo de obtener un buen rendimiento. Por ejemplo, la comparación de cadenas mediante la función *strcmp()* tiene una complejidad lineal respecto a la longitudad de las cadenas, es decir, una complejidad $O(n)$. Del mismo modo, la copia de cadenas también tiene una complejidad lineal, sin considerar la posibilidad de reservar memoria de manera dinámica. Por el contrario, la comparación de tipos de datos más simples, como los enteros, es muy eficiente y está soportado por instrucciones máquina.

7.3.2. Optimizando el tratamiento de cadenas

El uso de una clase *string* que abstraiga de la complejidad subyacente de la implementación de datos textuales es sin duda el mecanismo más práctico para trabajar con cadenas. Por ejemplo, en el caso de C++, el programador puede utilizar directamente la **clase *string***[6] de la biblioteca estándar. Dicha clase proporciona una gran cantidad de funcionalidad, estructurada en cinco bloques bien diferenciados:

- **Iteradores**, que permiten recorrer de manera eficiente el contenido de las instancias de la clase *string*. Un ejemplo es la función *begin()*, que devuelve un iterador al comienzo de la cadena.

[6] `http://www.cplusplus.com/reference/string/string/`

- **Capacidad**, que proporciona operaciones para obtener información sobre el tamaño de la cadena, redimensionarla e incluso modificar la cantidad de memoria reservada para la misma. Un ejemplo es la función *size()*, que devuelve la longitudad de la cadena.
- **Acceso**, que permite obtener el valor de caracteres concretos dentro de la cadena. Un ejemplo es la función *at()*, que devuelve el carácter de la posición especificada como parámetro.
- **Modificadores**, que proporcionan la funcionalidad necesaria para alterar el contenido de la cadena de texto. Un ejemplo es la función *swap()*, que intercambia el contenido de la cadena por otra especificada como parámetro.
- **Operadores de cadena**, que define operaciones auxiliares para tratar con cadenas. Un ejemplo es la función *compare()*, que permite comparar cadenas.

No obstante, tal y como se introdujo anteriormente, la abstracción proporcionada por este tipo de clases puede ocultar el **coste computacional** asociado a la misma y, en consecuencia, condicionar el rendimiento del juego. Por ejemplo, pasar una cadena de texto como parámetro a una función utilizando el estilo C, es decir, pasando un puntero al primer elemento de la cadena, es una operación muy eficiente. Sin embargo, pasar un objeto cadena puede ser ineficiente debido a la generación de copias mediante los constructores de copia, posiblemente provocando reserva dinámica de memoria y penalizando aún más el rendimiento de la aplicación.

Profiling strings!

El uso de herramientas de *profiling* para evaluar el impacto del uso y gestión de una implementación concreta de cadenas de texto puede ser esencial para mejorar el *frame rate* del juego.

Por este tipo de motivos, es bastante común que en el desarrollo profesional de videojuegos se evite el uso de clases *string* [42], especialmente en plataformas de propósito específico como las consolas de sobremesa. Si, por el contrario, en el desarrollo de un juego se hace uso de una clase de esta características, es importante considerar los siguientes aspectos:

- Realizar un estudio del impacto en complejidad espacial y temporal que tiene la adopción de una determinada implementación.
- Informar al equipo de desarrollo de la opción elegida para tener una perspectiva global de dicho impacto.
- Realizar un estudio de la opción elegida considerando aspectos específicos, como por ejemplo si todos los *buffers* son de sólo-lectura.

Como regla general, pase los objetos de tipo *string* como referencia y no como valor, ya que esto incurriría en una penalización debido al uso de constructores de copia.

En el ámbito del desarrollo de videojuegos, la **identificación de entidades** o elementos está estrechamente ligada con el uso de cadenas de texto. Desde un punto de vista general, la identificación unívoca de objetos en el mundo virtual de un juego es una parte esencial en el desarrollo como soporte a otro tipo de operaciones, como por ejemplo la localización de entidades en tiempo de ejecución por parte del motor de juegos. Así mismo, las entidades más van allá de los objetos virtuales. En realidad, también abarcan mallas poligonales, materiales, texturas, luces virtuales, animaciones, sonidos, etcétera.

En este contexto, las cadenas de texto representan la manera más natural de llevar a cabo dicha identificación y tratamiento. Otra opción podría consistir en manejar una **tabla con valores enteros** que sirviesen como identificadores de las distintas entidadas. Sin embargo, los valores enteros no permiten expresar valores semánticos, mientras que las cadenas de texto sí.

Debido a la importancia del tratamiento y manipulación de cadenas, las operaciones asociadas han de ser eficientes. Desafortunadamente, este criterio no se cumple en el caso de funciones como *strcmp()*. La solución ideal debería combinar la flexibilidad y poder descriptivo de las cadenas con la eficiencia de un tipo de datos primitivo, como los enteros. Este planteamiento es precisamente la base del esquema que se discute a continuación.

7.3.3. *Hashing* de cadenas

String id

En el ámbito profesional, el término *string id* se utiliza comúnmente para referirse a la cadena accesible mediante un determinado valor o código *hash*.

El uso de una estructura asociativa (ver sección 5.4) se puede utilizar para manejar de manera eficiente cadenas textuales. La idea se puede resumir en utilizar una estructura asociativa en la que la clave sea un valor numérico y la clase sea la propia cadena de texto. De este modo, la comparación entre los códigos *hash* de las cadenas, es decir, los enteros, es muy rápida.

Si las cadenas se almacenan en una **tabla *hash***, entonces el contenido original se puede recuperar a partir del código *hash*. Este esquema es especialmente útil en el proceso de depuración para mostrar el contenido textual, ya sea de manera directa por un dispositivo de visualización o mediante los típicos archivos de *log*.

Interning the string

El proceso que permite la generación de un identificador número a partir de una cadena de texto se suele denominar *interning*, ya que implica, además de realizar el *hashing*, almacenar la cadena en una tabla de visibilidad global.

Una de las principales cuestiones a considerar cuando se utilizan este tipo de planteamientos es el diseño de la función de *hashing*. En otras palabras, la función que traduce las claves, representadas normalmente mediante tipos de datos numéricos, en valores, representados por cadenas textuales en esta sección. El principal objetivo consiste en **evitar colisiones**, es decir, evitar que dos cadenas distintas tengan asociadas el mismo código o clave. Esta última situación se define como *perfect hashing* y existe una gran cantidad de información en la literatura[7].

Respecto a la implementación de un esquema basado en *hashing* de cadenas, uno de los aspectos clave consiste en decidir cuándo llamar a la función de *hashing*. Desde un punto de vista general, la mayoría de los motores de juegos permiten obtener el identificador asociado a una cadena en tiempo de ejecución. Sin embargo, es posible aplicar algunos trucos para optimizar dicha funcionalidad. Por ejemplo, es bastante común hacer uso de macros para obtener el identificador a partir de una cadena para su posterior uso en una sentencia *switch*.

También es importante tener en cuenta que el resultado de una función de *hashing* se suele almacenar en algún tipo de estructura de datos, en ocasiones global, con el objetivo de incrementar la eficiencia en consultas posteriores. Este planteamiento, parecido a un esquema de caché para las cadenas textuales, permite mejorar el rendimiento de la aplicación. El siguiente listado de código muestra una posible implementación que permite el manejo de cadenas de texto mediante una estructura de datos global.

Listado 7.14: Ejemplo de hashing de cadenas.

```
1 static StringId sid_hola = internString("hola");
2 static StringId sid_mundo = internString("mundo");
3
4 // ...
```

[7] Se recomienda la visualización del curso *Introduction to Algorithms* del MIT, en concreto las clases 7 y 8, disponible en la web.

```
5
6  void funcion (StringId id) {
7    // Más eficiente que...
8    // if (id == internString ("hola"))
9    if (id == sid_hola)
10     // ...
11   else if (id == sid_mundo)
12     // ...
13 }
```


7.4. Configuración del motor

Debido a la complejidad inherente a un motor de juegos, existe una gran variedad de variables de configuración que se utilizan para especificar el comportamiento de determinadas partes o ajustar cuestiones específicas. Desde un punto de vista general, en la configuración del motor se pueden distinguir dos grandes bloques:

- **Externa**, es decir, la configuración vinculada al juego, no tanto al motor, que el usuario puede modificar directamente. Ejemplos representativos pueden ser la configuración del nivel de dificultad de un juego, los ajustes más básicos de sonido, como el control de volumen, o incluso el nivel de calidad gráfica.
- **Interna**, es decir, aquellos aspectos de configuración utilizados por los desarrolladores para realizar ajustes que afecten directamente al comportamiento del juego. Ejemplos representativos pueden ser la cantidad de tiempo que el personaje principal puede estar corriendo sin cansarse o la vitalidad del mismo. Este tipo de parámetros permiten completar el proceso de depuración permanecen ocultos al usuario del juego.

Tricks

En muchos juegos es bastante común encontrar trucos que permiten modificar significativamente algunos aspectos internos del juego, como por ejemplo la capacidad de desplazamiento del personaje principal. Tradicionalmente, la activación de trucos se ha realizado mediante combinaciones de teclas.

7.4.1. Esquemas típicos de configuración

Las variables de configuración se pueden definir de manera trivial mediante el uso de variables globales o variables miembro de una clase que implemente el patrón *singleton*. Sin embargo, idealmente debería ser posible modificar dichas variables de configuración sin necesidad de modificar el código fuente y, por lo tanto, volver a compilar para generar un ejecutable.

Normalmente, las variables o parámetros de configuración residen en algún tipo de dispositivo de almacenamiento externo, como por ejemplo un disco duro o una tarjeta de memoria. De este modo, es posible almacenar y recuperar la información asociada a la configuración de una manera práctica y directa. A continuación se discute brevemente los distintas aproximaciones más utilizadas en el desarrollo de videojuegos.

En primer lugar, uno de los esquemás típicos de recuperación y almacenamiento de información está representado por los **ficheros de configuración**. La principal ventaja de este enfoque es que son perfectamente legibles ya que suelen especificar de manera explícita la variable de configuración a modificar y el valor asociada a la misma. En general, cada motor de juegos mantiene su propio convenio aunque, normalmente, todos se basan en una secuencia de pares clave-valor. Por ejemplo, Ogre3D utiliza la siguiente nomenclatura:

```
Render System=OpenGL Rendering Subsystem

[OpenGL Rendering Subsystem]
```

```
Display Frequency=56 MHz
FSAA=0
Full Screen=No
RTT Preferred Mode=FBO
VSync=No
Video Mode= 800 x  600
sRGB Gamma Conversion=No
```

Dentro del ámbito de los ficheros de configuración, los **archivos XML** representan otra posibilidad para establecer los parámetros de un motor de juegos. En este caso, es necesario llevar a cabo un proceso de *parsing* para obtener la información asociada a dichos parámetros.

El lenguaje XML

eXtensible Markup Language es un metalenguaje extensible basado en etiquetas que permite la definición de lenguajes específicos de un determinado dominio. Debido a su popularidad, existen multitud de herramientas que permiten el tratamiento y la generación de archivos bajo este formato.

Tradicionalmente, las plataformas de juego que sufrían restricciones de memoria, debido a limitaciones en su capacidad, hacían uso de un **formato binario** para almacenar la información asociada a la configuración. Típicamente, dicha información se almacenaba en tarjetas de memoria externas que permitían salvar las partidas guardadas y la configuración proporcionada por el usuario de un determinado juego.

Este planteamiento tiene como principal ventaja la eficiencia en el almacenamiento de información. Actualmente, y debido en parte a la convergencia de la consola de sobremesa a estaciones de juegos con servicios más generales, la limitación de almacenamiento en memoria permanente no es un problema, normalmente. De hecho, la mayoría de consolas de nueva generación vienen con discos duros integrados.

Los propios **registros** del sistema operativo también se pueden utilizar como soporte al almacenamiento de parámetros de configuración. Por ejemplo, los sistemas operativos de la familia de *Microsoft Windows*™ proporcionan una base de datos global implementada mediante una estructura de árbol. Los nodos internos de dicha estructura actúan como directorios mientras que los nodos hoja representan pares clave-valor.

También es posible utilizar la **línea de órdenes** o incluso la definición de variables de entorno para llevar a cabo la configuración de un motor de juegos.

Finalmente, es importante reflexionar sobre el futuro de los esquemas de almacenamiento. Actualmente, es bastante común encontrar servicios que permitan el almacenamiento de partidas e incluso de preferencias del usuario en la red, haciendo uso de **servidores en Internet** normalmente gestionados por la propia compañía de juegos. Este tipo de aproximaciones tienen la ventaja de la redundancia de datos, sacrificando el control de los datos por parte del usuario.

7.4.2. Caso de estudio. Esquemas de definición.

Una técnica bastante común que está directamente relacionada con la configuración de un motor de juegos reside en el uso de esquemas de definición de datos. Básicamente, la idea general reside en hacer uso de algún tipo de **lenguaje de programación declarativo**, como por ejemplo LISP, para llevar a cabo la definición de datos que permitan su posterior exportación a otros lenguajes de programación. Este tipo de lenguajes son muy flexibles y proporcionan una gran potencia a la hora de definir estructuras de datos.

LISP

Junto con Fortran, LISP es actualmente uno de los lenguajes de programación más antiguos que se siguen utilizando comercialmente. Sus orígenes están fuertemente ligados con la Inteligencia Artificial y fue pionero de ideas fundamentales en el ámbito de la computación, como los árboles, la gestión automática del almacenamiento o el *tipado* dinámico.

A continuación se muestra un ejemplo real del videojuego *Uncharted: Drake's Fortune*, desarrollado por *Naughty Dog* para la consola de sobremesa *Playstation 3*™, y discutido en profundidad en [42]. En dicho ejemplo se define una estructura básica para almacenar las propiedades de una animación y dos instancias vinculadas a un tipo particular de animación, utilizando para ello un lenguaje propietario.

Listado 7.15: Definición de una animación básica [42]

```
;; Estructura básica de animación.
(define simple-animation ()
  (
   (name               string)
   (speed              float     :default 1.0)
   (fade-in-seconds    float     :default 0.25)
   (fade-out-seconds   float     :default 0.25)
  )
)

;; Instancia específica para andar.
(define-export anim-walk
  (new simple-animation
       :name "walk"
       :speed 1.0
  )
)
;; Instancia específica para andar rápido.
(define-export anim-walk-fast
  (new simple-animation
       :name "walk-fast"
       :speed 2.0
  )
)
```

Evidentemente, la consecuencia directa de definir un lenguaje propio de definición de datos implica el desarrollo de un **compilador específico** que sea capaz de generar de manera automática código fuente a partir de las definiciones creadas. En el caso de la definición básica de animación planteada en el anterior listado de código, la salida de dicho compilador podría ser similar a la mostrada a continuación.

Listado 7.16: Estructura generada por el compilador.

```
// Código generado a partir del compilador.
// Estructura básica de animación.

struct SimpleAnimation {
  const char*  m_name;
  float        m_speed;
  float        m_fadeInSeconds;
  float        m_fadeOutSeconds;
};
```

7.5. Fundamentos básicos de concurrencia

7.5.1. El concepto de hilo

Elementos de un hilo

Un hilo está compuesto por un identificador único de hilo, un contador de programa, un conjunto de registros y una pila.

Los sistemas operativos modernos se basan en el principio de **multiprogramación**, es decir, en la posibilidad de manejar distintos hilos de ejecución de manera simultánea con el objetivo de paralelizar el código e incrementar el rendimiento de la aplicación.

Esta idea también se plasma a nivel de lenguaje de programación. Algunos ejemplos representativos son las APIs de las bibliotecas de hilos *Pthread*, *Win32* o *Java*. Incluso existen bibliotecas de gestión de hilos que se enmarcan en capas software situadas sobre la capa del sistema operativo, con el objetivo de independizar el modelo de programación del propio sistema operativo subyacente.

Este último planteamiento es uno de los más utilizados en el desarrollo de videojuegos con el objetivo de evitar posibles problemas a la hora de portarlos a diversas plataformas. Recuerde que en la sección 1.2.3, donde se discutió la arquitectura general de un motor de juegos, se justificaba la necesidad de incluir una **capa independiente de la plataforma** con el objetivo de garantizar la portabilidad del motor. Evidentemente, la inclusión de una nueva capa software degrada el rendimiento final de la aplicación.

Es importante considerar que, a diferencia de un proceso, los hilos que pertenecen a un mismo proceso comparten la sección de código, la sección de datos y otros recursos proporcionados por el sistema operativo (ver figura 7.12), como los manejadores de los archivos abiertos. Precisamente, esta diferencia con respecto a un proceso es lo que supone su principal ventaja a la hora de utilizar un elemento u otro.

Informalmente, un hilo se puede definir como un *proceso ligero* que tiene la misma funcionalidad que un *proceso pesado*, es decir, los mismo estados. Por ejemplo, si un hilo abre un fichero, éste estará disponible para el resto de hilos de una tarea. Las **ventajas de la programación multihilo** se pueden resumir en las tres siguientes:

Estados de un hilo

Un hilo mantiene los mismos estados internos que un proceso: *nuevo*, *ejecución*, *espera*, *preparado* y *terminado*.

- **Capacidad de respuesta**, ya que el uso de múltiples hilos proporciona un enfoque muy flexible. Así, es posible que un hilo se encuentra atendiendo una petición de E/S mientras otro continúa con la ejecución de otra funcionalidad distinta. Además, es posible plantear un esquema basado en el paralelismo no bloqueante en llamadas al sistema, es decir, un esquema basado en el bloqueo de un hilo a nivel individual.
- **Compartición de recursos**, posibilitando que varios hilos manejen el mismo espacio de direcciones.
- **Eficacia**, ya que tanto la creación, el cambio de contexto, la destrucción y la liberación de hilos es un orden de magnitud más rápida que en el caso de los procesos pesados. Recuerde que las operaciones más costosas implican el manejo de operaciones de E/S. Por otra parte, el uso de este tipo de programación en arquitecturas de varios procesadores (o núcleos) incrementa enormemente el rendimiento de la aplicación.

7.5.2. El problema de la sección crítica

El segmento de código en el que un proceso puede modificar variables compartidas con otros procesos se denomina sección crítica. Un ejemplo típico en el ámbito de la orientación a objetos son las variables miembro de una clase, suponiendo que las instancias de la misma pueden atender múltiples peticiones de manera simultánea mediante distintos hilos de control.

Para **evitar inconsistencias**, una de las ideas que se plantean es que cuando un hilo está ejecutando su sección crítica, ningún otro hilo puede acceder a dicha sección crítica. Así, si un objeto puede manera múltiples peticiones, no debería ser posible que un cliente modifique el estado de dicha instancia mientras otro intenta leerlo.

Condición de carrera

Si no se protege adecuadamente la sección crítica, distintas ejecuciones de un mismo código pueden generar distintos resultados, generando así una condición de carrera.

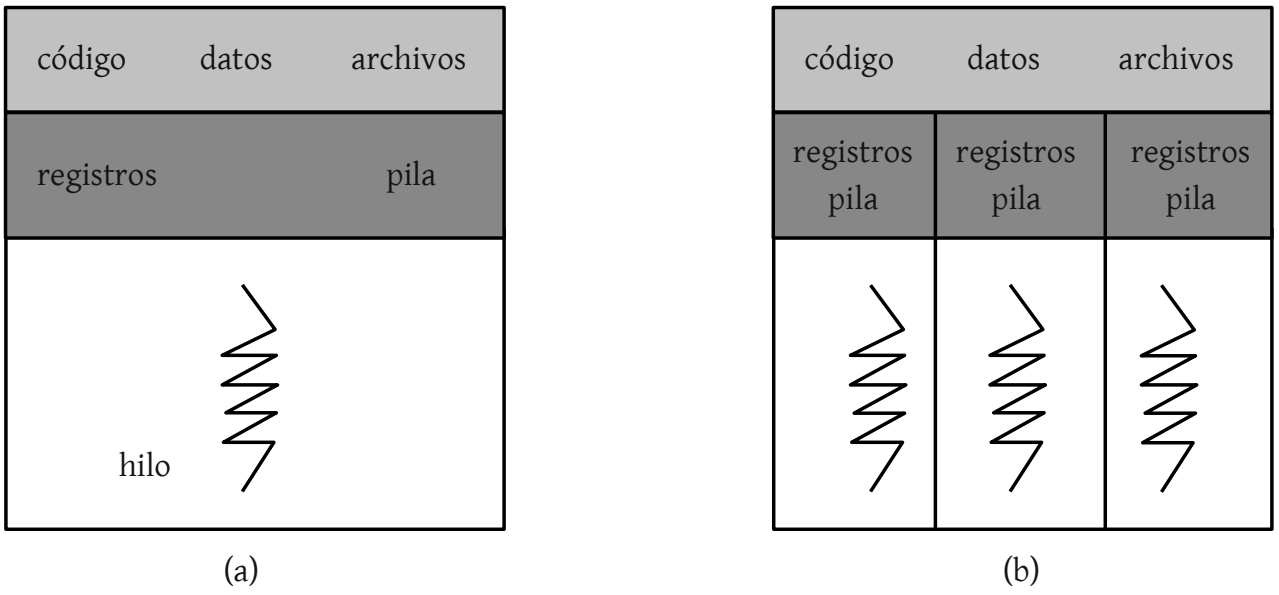


Figura 7.12: Esquema gráfico de los modelos de programación monohilo (a) y multihilo (b).

Figura 7.13: Estructura general del código vinculado a la sección crítica.

El **problema de la sección crítica** consiste en diseñar algún tipo de solución para garantizar que los elementos involucrados puedan operar sin generar ningún tipo de inconsistencia. Una posible estructura para abordar esta problemática se plantea en la figura 7.13, en la que el código se divide en las siguientes secciones:

- **Sección de entrada**, en la que se solicita el acceso a la sección crítica.
- **Sección crítica**, en la que se realiza la modificación efectiva de los datos compartidos.
- **Sección de salida**, en la que típicamente se hará explícita la salida de la sección crítica.
- **Sección restante**, que comprende el resto del código fuente.

Nombre	Internet Communications Engine (ICE)
Definido por	ZeroC Inc. (`http://www.zeroc.com`)
Documentación	`http://doc.zeroc.com/display/Doc/Home`
Lenguajes	C++, Java, C#, Visual Basic, Python, PHP, Ruby
Plataformas	Windows, Windows CE, Unix, GNU/Linux, *BSD OSX, Symbian OS, J2RE 1.4 o superior, J2ME
Destacado	APIs claras y bien diseñadas Conjunto de servicios muy cohesionados Despliegue, persistencia, cifrado...
Descargas	`http://zeroc.com/download.html`

Cuadro 7.2: ZeroC ICE. Resumen de características.

7.6. La biblioteca de hilos de ICE

7.6.1. Internet Communication Engine

ICE (Internet Communication Engine) es un *middleware* de comunicaciones orientado a objetos, es decir, ICE proporciona herramientas, APIs, y soporte de bibliotecas para construir aplicaciones distribuidas cliente-servidor orientadas a objetos (ver figura 7.14).

Una aplicación ICE se puede usar en entornos heterogéneos. Los clientes y los servidores pueden escribirse en diferentes lenguajes de programación, pueden ejecutarse en distintos sistemas operativos y en distintas arquitecturas, y pueden comunicarse empleando diferentes tecnologías de red. La tabla 7.2 resume las principales características de ICE.

Los principales **objetivos de diseño** de ICE son los siguientes:

- *Middleware* listo para usarse en sistemas heterogéneos.
- Proveee un conjunto completo de características que soporten el desarrollo de aplicaciones distribuidas reales en un amplio rango de dominios.
- Es fácil de aprender y de usar.
- Proporciona una implementación eficiente en ancho de banda, uso de memoria y CPU.
- Implementación basada en la seguridad.

Para instalar ICE en sistemas operativos Debian GNU/Linux, ejecute los siguientes comandos:

```
$ sudo apt-get update
$ sudo apt-get install zeroc-ice35
```

En el módulo 4, *Desarrollo de Componentes*, se estudiarán los aspectos básicos de este potente *middleware* para construir aplicaciones distribuidas. Sin embargo, en la presente sección se estudiará la **biblioteca de hilos** que proporciona ICE para dar soporte a la gestión de la concurrencia y al desarrollo multihilo cuando se utiliza el lenguaje de programación C++.

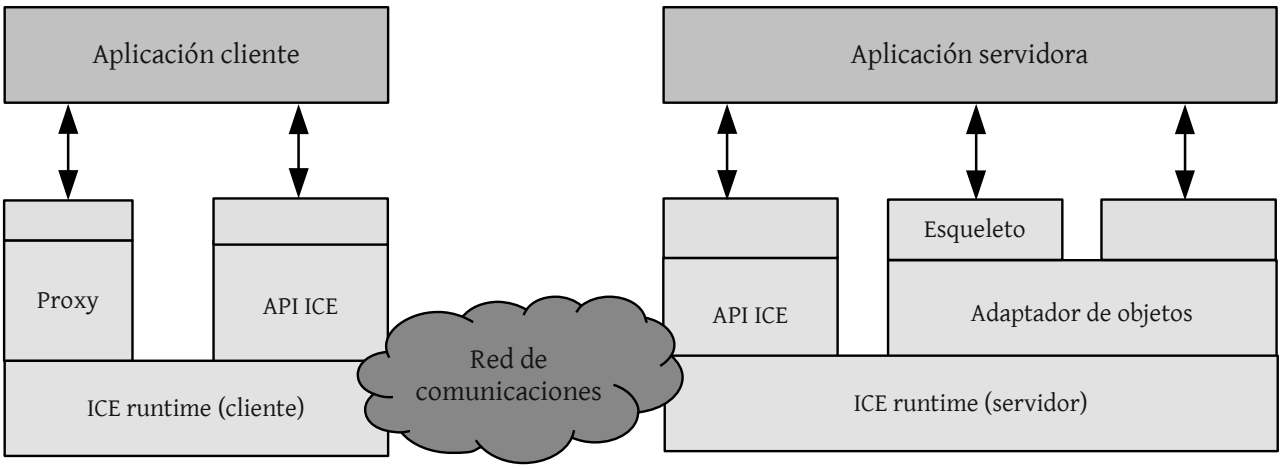


Figura 7.14: Arquitectura general de una aplicación distribuida desarrollada con el *middleware* ZeroC ICE.

El objetivo principal de esta biblioteca es abstraer al desarrollador de las capas inferiores e independizar el desarrollo de la aplicación del sistema operativo y de la plataforma hardware subyacentes. De este modo, la portabilidad de las aplicaciones desarrolladas con esta biblioteca se garantiza, evitando posibles problemas de incompatibilidad entre sistemas operativos.

Thread-safety

ICE maneja las peticiones a los servidores mediante un *pool* de hilos con el objetivo de incrementar el rendimiento de la aplicación. El desarrollador es el responsable de gestionar el acceso concurrente a los datos.

7.6.2. Manejo de hilos

ICE proporciona distintas utilidades para la gestión de la concurrencia y el manejo de hilos. Respecto a este último aspecto, ICE proporciona una **abstracción** muy sencilla para el manejo de hilos con el objetivo de explotar el paralelismo mediante la creación de hilos dedicados. Por ejemplo, sería posible crear un hilo específico que atienda las peticiones de una interfaz gráfica o crear una serie de hilos encargados de tratar con aquellas operaciones que requieren una gran cantidad de tiempo, ejecutándolas en segundo plano.

En este contexto, ICE proporciona una abstracción de hilo muy sencilla que posibilita el desarrollo de aplicaciones multihilo altamente portables e independientes de la plataforma de hilos nativa. Esta abstracción está representada por la clase **IceUtil::Thread**.

Como se puede apreciar en el siguiente listado de código, *Thread* es una clase abstracta con una función virtual pura denominada **run()**. El desarrollador ha de implementar esta función para poder crear un hilo, de manera que *run()* se convierta en el punto de inicio de la ejecución de dicho hilo. Note que no es posible arrojar excepciones desde esta función. El núcleo de ejecución de ICE instala un manejador de excepciones que llama a la función *::std::terminate()* si se arroja alguna excepción.

El resto de funciones de *Thread* son las siguientes:

- **start()**, cuya responsabilidad es arrancar al hilo y llamar a la función *run()*. Es posible especificar el tamaño en *bytes* de la pila del nuevo hilo, así como un valor de prioridad. El valor de retorno de *start()* es un objeto de tipo *ThreadControl*, el cual se discutirá más adelante.
- **getThreadControl()**, que devuelve el objeto de la clase *ThreadControl* asociado al hilo.

Listado 7.17: La clase IceUtil::Thread

```
1  class Thread :virtual public Shared {
2   public:
3
4    // Función a implementar por el desarrollador.
5    virtual void run () = 0;
6
7    ThreadControl start (size_t stBytes = 0);
8    ThreadControl start (size_t stBytes, int priority);
9    ThreadControl getThreadControl () const;
10   bool isAlive () const;
11
12   bool operator== (const Thread&) const;
13   bool operator!= (const Thread&) const;
14   bool operator<  (const Thread&) const;
15 };
16
17 typedef Handle<Thread> ThreadPtr;
```

- **id()**, que devuelve el identificador único asociado al hilo. Este valor dependerá del soporte de hilos nativo (por ejemplo, POSIX *pthreads*).
- **isAlive()**, que permite conocer si un hilo está en ejecución, es decir, si ya se llamó a *start()* y la función *run()* no terminó de ejecutarse.
- La **sobrecarga de los operadores** de comparación permiten hacer uso de hilos en contenedores de STL que mantienen relaciones de orden.

Para mostrar cómo llevar a cabo la implementación de un hilo específico a partir de la biblioteca de ICE se usará como ejemplo uno de los problemas clásicos de sincronización: el problema de los **filósofos comensales**.

Básicamente, los filósofos se encuentran comiendo o pensando. Todos comparten una mesa redonda con cinco sillas, una para cada filósofo. Cada filósofo tiene un plato individual con arroz y en la mesa sólo hay cinco palillos, de manera que cada filósofo tiene un palillo a su izquierda y otro a su derecha.

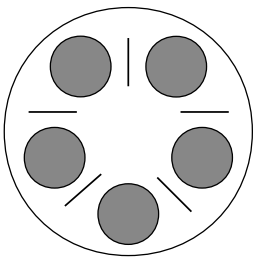


Figura 7.15: Abstracción gráfica del problema de los filósofos comensales, donde cinco filósofos piensan y comparten cinco palillos para comer.

Listado 7.18: La clase FilosofoThread

```
1  #ifndef __FILOSOFO__
2  #define __FILOSOFO__
3
4  #include <iostream>
5  #include <IceUtil/Thread.h>
6  #include <Palillo.h>
7
8  #define MAX_COMER 3
9  #define MAX_PENSAR 7
10
11 using namespace std;
12
13 class FilosofoThread : public IceUtil::Thread {
14
15  public:
16   FilosofoThread (const int& id, Palillo* izq, Palillo *der);
17
18   virtual void run ();
19
20  private:
```

```
21   void coger_palillos ();
22   void dejar_palillos ();
23   void comer () const;
24   void pensar () const;
25
26   int _id;
27   Palillo *_pIzq, *_pDer;
28 };
29
30 #endif
```


Sincronización

El problema de los filósofos comensales es uno de los problemas clásicos de sincronización y existen muchas modificaciones del mismo que permiten asimilar mejor los conceptos teóricos de la programación concurrente.

Cuando un filósofo piensa, entonces se abstrae del mundo y no se relaciona con ningún otro filósofo. Cuando tiene hambre, entonces intenta coger a los palillos que tiene a su izquierda y a su derecha (necesita ambos). Naturalmente, un filósofo no puede quitarle un palillo a otro filósofo y sólo puede comer cuando ha cogido los dos palillos. Cuando un filósofo termina de comer, deja los palillos y se pone a pensar.

La solución que se discutirá en esta sección se basa en implementar el filósofo como un hilo independiente. Para ello, se crea la clase *FilosofoThread* que se expone en el anterior listado de código.

La implementación de la función *run()* es trivial a partir de la descripción del enunciado del problema.

Listado 7.19: La función FilosofoThread::run()

```
1  void
2  FilosofoThread::run ()
3  {
4    while (true) {
5      coger_palillos();
6      comer();
7      dejar_palillos();
8      pensar();
9    }
10 }
```

El problema de concurrencia viene determinado por el acceso de los filósofos a los palillos, los cuales representan la sección crítica asociada a cada uno de los hilos que implementan la vida de un filósofo. En otras palabras, es necesario establecer algún tipo de mecanismo de sincronización para garantizar que dos filósofos no cogen un mismo palillo de manera simultánea. Antes de abordar esta problemática, se mostrará cómo lanzar los hilos que representan a los cinco filósofos.

El siguiente listado de código muestra el código básico necesario para lanzar los filósofos (hilos). Note cómo en la línea 25 se llama a la función *start()* de *Thread* para comenzar la ejecución del mismo. Los objetos de tipo *ThreadControl* devueltos se almacenan en un vector para, posteriormente, *unir* los hilos creados. Para ello, se hace uso de la función *join()* de la clase *ThreadControl*, tal y como se muestra en la línea 31.

7.6.3. Exclusión mutua básica

Locking and unlocking

Típicamente, las operaciones para adquirir y liberar un cerrojo se denominan *lock()* y *unlock()*, respectivamente. La metáfora de cerrojo representa perfectamente la adquisición y liberación de recursos compartidos.

El problema de los filósofos plantea la necesidad de algún tipo de mecanismo de sincronización básico para garantizar el acceso exclusivo sobre cada uno de los palillos. La opción más directa consiste en asociar un **cerrojo** a cada uno de los palillos de manera individual. Así, si un filósofo intenta coger un palillo que está libre, entonces lo cogerá adquiriendo el cerrojo, es decir, *cerrándolo*. Si el palillo está ocupado, entonces el filósofo esperará hasta que esté libre.

Listado 7.20: Creación y control de hilos con ICE

```
#include <IceUtil/Thread.h>
#include <vector>
#include <Palillo.h>
#include <Filosofo.h>

#define NUM 5

int main
(int argc, char *argv[]) {
  std::vector<Palillo*> palillos;
  std::vector<IceUtil::ThreadControl> threads;
  int i;

  // Se instancian los palillos.
  for (i = 0; i < NUM; i++)
    palillos.push_back(new Palillo);

  // Se instancian los filósofos.
  for (i = 0; i < NUM; i++) {
    // Cada filósofo conoce los palillos
    // que tiene a su izda y derecha.
    IceUtil::ThreadPtr t =
      new FilosofoThread(i, palillos[i], palillos[(i + 1) % NUM]);
    // start sobre hilo devuelve un objeto ThreadControl.
    threads.push_back(t->start());
  }

  // 'Unión' de los hilos creados.
  std::vector<IceUtil::ThreadControl>::iterator it;
  for (it = threads.begin(); it != threads.end(); ++it)
    it->join();

  return 0;
}
```

ICE proporciona la clase **Mutex** para modelar esta problemática de una forma sencilla y directa.

Las funciones miembro más importantes de esta clase son las que permiten adquirir y liberar el cerrojo:

- **lock()**, que intenta adquirir el cerrojo. Si éste ya estaba cerrado, entonces el hilo que invocó a la función se suspende hasta que el cerrojo quede libre. La llamada a dicha función retorna cuando el hilo ha adquirido el cerrojo.
- **tryLock()**, que intenta adquirir el cerrojo. A diferencia de *lock()*, si el cerrojo está cerrado, la función devuelve *false*. En caso contrario, devuelve *true* con el cerrojo cerrado.
- **unlock()**, que libera el cerrojo.

Figura 7.16: Los cerrojos o *mutex* representan un mecanismo de sincronización muy básico pero ampliamente utilizado.

Listado 7.21: La clase IceUtil::Mutex

```
class Mutex {
 public:
  Mutex  ();
  Mutex  (MutexProtocol p);
  ~Mutex ();

  void lock    () const;
  bool tryLock () const;
  void unlock  () const;

```

```
11   typedef LockT<Mutex> Lock;
12   typedef TryLockT<Mutex> TryLock;
13 };
```

Es importante considerar que la clase *Mutex* proporciona un mecanismo de exclusión mutua básico y no recursivo, es decir, no se debe llamar a *lock()* más de una vez desde un hilo, ya que esto provocará un comportamiento inesperado. Del mismo modo, no se debería llamar a *unlock()* a menos que un hilo haya adquirido previamente el cerrojo mediante *lock()*. En la siguiente sección se estudiará otro mecanismo de sincronización que mantiene una semántica recursiva.

La clase *Mutex* se puede utilizar para gestionar el acceso concurrente a los palillos. En la solución planteada a continuación, un palillo es simplemente una especialización de *IceUtil::Mutex* con el objetivo de incrementar la semántica de dicha solución.

Riesgo de interbloqueo

Si todos los filósofos cogen al mismo tiempo el palillo que está a su izquierda se producirá un interbloqueo ya que la solución planteada no podría avanzar hasta que un filósofo coja ambos palillos.

Listado 7.22: La clase Palillo

```
1 #ifndef __PALILLO__
2 #define __PALILLO__
3
4 #include <IceUtil/Mutex.h>
5
6 class Palillo : public IceUtil::Mutex {
7 };
8
9 #endif
```

Para que los filósofos puedan utilizar los palillos, habrá que utilizar la funcionalidad previamente discutida, es decir, las funciones *lock()* y *unlock()*, en las funciones *coger_palillos()* y *dejar_palillos()*. La solución planteada garantiza que no se ejecutarán dos llamadas a *lock()* sobre un palillo por parte de un mismo hilo, ni tampoco una llamada sobre *unlock()* si previamente no se adquirió el palillo.

Listado 7.23: Acceso concurrente a los palillos

```
1  void
2  FilosofoThread::coger_palillos ()
3  {
4    _pIzq->lock();
5    _pDer->lock();
6  }
7
8  void
9  FilosofoThread::dejar_palillos ()
10 {
11   _pIzq->unlock();
12   _pDer->unlock();
13 }
```

La solución planteada es poco flexible debido a que los filósofos están inactivos durante el periodo de tiempo que pasa desde que dejan de pensar hasta que cogen los dos palillos. Una posible variación a la solución planteada hubiera sido continuar pensando (al menos hasta un número máximo de ocasiones) si los palillos están ocupados. En esta variación se podría utilizar la función *tryLock()* para modelar dicha problemática.

Gestión del *deadlock*

Aunque existen diversos esquemas para evitar y recuperarse de un interbloqueo, los sistemas operativos modernos suelen optar por asumir que nunca se producirán, delegando en el programador la implementación de soluciones seguras.

Evitando interbloqueos

El uso de las funciones *lock()* y *unlock()* puede generar problemas importantes si, por alguna situación no controlada, un cerrojo previamente adquirido con *lock()* no se libera posteriormente con una llamada a *unlock()*. El siguiente listado de código muestra esta problemática.

Listado 7.24: Potencial interbloqueo

```
1  #include <IceUtil/Mutex.h>
2
3  class Test {
4  public:
5    void mi_funcion () {
6      _mutex.lock();
7      for (int i = 0; i < 5; i++)
8        if (i == 3) return; // Generará un problema...
9      _mutex.unlock();
10   }
11
12 private:
13   IceUtil::Mutex _mutex;
14 };
15
16 int main (int argc, char *argv[]) {
17   Test t;
18   t.mi_funcion();
19   return 0;
20 }
```

En la línea 8, la sentencia *return* implica abandonar *mi_funcion()* sin haber liberado el cerrojo previamente adquirido en la línea 6. En este contexto, es bastante probable que se genere una potencial situación de interbloqueo que paralizaría la ejecución del programa. La generación de excepciones no controladas representa otro caso representativo de esta problemática.

Para evitar este tipo de problemas, la clase *Mutex* proporciona las definiciones de tipo *Lock* y *TryLock*, que representan plantillas muy sencillas compuestas de un constructor en el que se llama a *lock()* y *tryLock()*, respectivamente. En el destructor se llama a *unlock()* si el cerrojo fue previamente adquirido cuando la plantilla quede fuera de ámbito. En el ejemplo anterior, sería posible garantizar la liberación del cerrojo al ejecutar *return*, ya que quedaría fuera del alcance de la función. El siguiente listado de código muestra la modificación realizada para evitar interbloqueos.

Listado 7.25: Evitando interbloqueos con Lock

```
1  #include <IceUtil/Mutex.h>
2
3  class Test {
4  public:
5    void mi_funcion () {
6      IceUtil::Mutex::Lock lock(_mutex);
7      for (int i = 0; i < 5; i++)
8        if (i == 3) return; // Ningún problema...
9    } // El destructor de lock libera el cerrojo.
10
11 private:
12   IceUtil::Mutex _mutex;
13 };
```

Es recomendable usar siempre *Lock* y *TryLock* en lugar de las funciones *lock()* y *unlock* para facilitar el entendimiento y la mantenibilidad del código.

7.6.4. Flexibilizando el concepto de *mutex*

No olvides...

Liberar un cerrojo sólo si fue previamente adquirido y llamar a *unlock()* tantas veces como a *lock()* para que el cerrojo quede disponible para otro hilo.

Además de proporcionar cerrojos con una semántica no recursiva, ICE también proporciona la clase *IceUtil::RecMutex* con el objetivo de que el desarrollador pueda manejar cerrojos *recursivos*. La interfaz de esta nueva clase es exactamente igual que la clase *IceUtil::Mutex*.

Sin embargo, existe una diferencia fundamental entre ambas. Internamente, el cerrojo recursivo está implementado con un contador inicializado a cero. Cada llamada a *lock()* incrementa el contador, mientras que cada llamada a *unlock()* lo decrementa. El cerrojo estará disponible para otro hilo cuando el contador alcance el valor de cero.

Listado 7.26: La clase IceUtil::RecMutex

```
1  class RecMutex {
2   public:
3    RecMutex  ();
4    RecMutex  (MutexProtocol p);
5    ~RecMutex ();
6
7    void lock    () const;
8    bool tryLock () const;
9    void unlock  () const;
10
11   typedef LockT<RecMutex> Lock;
12   typedef TryLockT<RecMutex> TryLock;
13 };
```

7.6.5. Introduciendo monitores

Tanto la clase *Mutex* como la clase *MutexRec* son mecanismos de sincronización básicos que permiten que sólo un hilo esté activo, en un instante de tiempo, dentro de la sección crítica. En otras palabras, para que un hilo pueda acceder a la sección crítica, otro ha de abandonarla. Esto implica que, cuando se usan cerrojos, no resulta posible suspender un hilo dentro de la sección crítica para, posteriormente, despertarlo cuando se cumpla una determinada condición.

Para tratar este tipo de problemas, la biblioteca de hilos de ICE proporciona la **clase *Monitor***. En esencia, un monitor es un mecanismo de sincronización de más alto nivel que, al igual que un cerrojo, protege la sección crítica y garantiza que solamente pueda existir un hilo activo dentro de la misma. Sin embargo, un monitor permite suspender un hilo dentro de la sección crítica posibilitando que otro hilo pueda acceder a la misma. Este segundo hilo puede abandonar el monitor, liberándolo, o suspenderse dentro del monitor. De cualquier modo, el hilo original se despierta y continua su ejecución dentro del monitor. Este esquema es escalable a múltiples hilos, es decir, varios hilos pueden suspenderse dentro de un monitor.

Solución de alto nivel

Las soluciones de alto nivel permiten que el desarrollador tenga más flexibilidad a la hora de solucionar un problema. Este planteamiento se aplica perfectamente al uso de monitores.

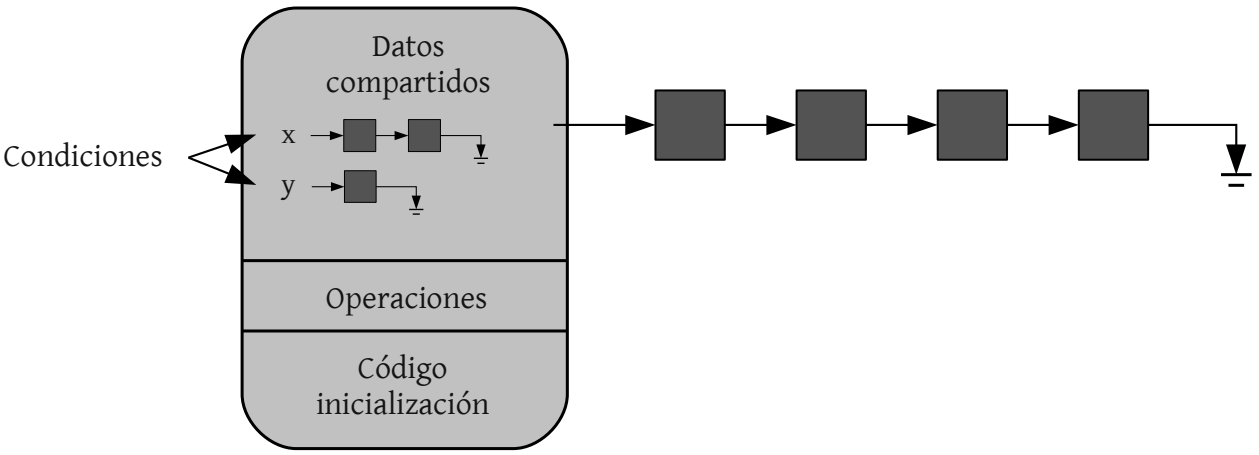


Figura 7.17: Representación gráfica del concepto de monitor.

Desde un punto de vista general, los monitores proporcionan un mecanismo de sincronización más flexible que los cerrojos, ya que es posible que un hilo compruebe una condición y, si ésta es falsa, el hilo se pause. Si otro hilo cambia dicha condición, entonces el hilo original continúa su ejecución.

El siguiente listado de código muestra la declaración de la clase **IceUtil::Monitor**. Note que se trata de una clase que hace uso de plantillas y que requiere como parámetro bien *Mutex* o *RecMutex*, en función de si el monitor mantendrá una semántica no recursiva o recursiva, respectivamente.

Listado 7.27: La clase IceUtil::Monitor

```
1  template <class T>
2  class Monitor {
3   public:
4    void lock    () const;
5    void unlock  () const;
6    bool tryLock () const;
7
8    void wait () const;
9    bool timedWait (const Time&) const;
10   void notify ();
11   void notifyAll ();
12
13   typedef LockT<Monitor<T> > Lock;
14   typedef TryLockT<Monitor<T> > TryLock;
15 };
```

Las funciones miembro de esta clase son las siguientes:

- **lock()**, que intenta adquirir el monitor. Si éste ya estaba cerrado, entonces el hilo que la invoca se suspende hasta que el monitor quede disponible. La llamada retorna con el monitor cerrado.
- **tryLock()**, que intenta adquirir el monitor. Si está disponible, la llamada devuelve *true* con el monitor cerrado. Si éste ya estaba cerrado antes de relizar la llamada, la función devuelve *false*.
- **unlock()**, que libera el monitor. Si existen hilos bloqueados por el mismo, entonces uno de ellos se despertará y cerrará de nuevo el monitor.

No olvides...

Comprobar la condición asociada al uso de *wait* siempre que se retorne de una llamada a la misma.

- **wait()**, que suspende al hilo que invoca a la función y, de manera simultánea, libera el monitor. Un hilo suspendido por *wait()* se puede despertar por otro hilo que invoque a la función *notify()* o *notifyAll()*. Cuando la llamada retorna, el hilo suspendido continúa su ejecución con el monitor cerrado.
- **timedWait()**, que suspende al hilo que invoca a la función hasta un tiempo especificado por parámetro. Si otro hilo invoca a *notify()* o *notifyAll()* despertando al hilo suspendido antes de que el *timeout* expire, la función devuelve *true* y el hilo suspendido resume su ejecución con el monitor cerrado. En otro caso, es decir, si el *timeout* expira, *timedWait()* devuelve *false*.
- **notify()**, que despierta a un único hilo suspendido debido a una invocación sobre *wait()* o *timedWait()*. Si no existiera ningún hilo suspendido, entonces la invocación sobre *notify()* se *pierde*. Llevar a cabo una notificación no implica que otro hilo reanude su ejecución inmediatamente. En realidad, esto ocurriría cuando el hilo que invoca a *wait()* o *timedWait()* libera el monitor.
- **notifyAll()**, que despierta a todos los hilos suspendidos por *wait()* o *timedWait()*. El resto del comportamiento derivado de invocar a esta función es idéntico a *notify()*.

Ejemplo de uso de monitores

Imagine que desea modelar una situación en un videojuego en la que se controlen los **recursos disponibles** para acometer una determinada tarea. Estos recursos se pueden manipular desde el punto de vista de la generación o producción y desde el punto de vista de la destrucción o consumición. En otras palabras, el problema clásico del productor/consumidor.

Por ejemplo, imagine un juego de acción de tercera persona en el que el personaje es capaz de **acumular *slots*** para desplegar algún tipo de poder especial. Inicialmente, el personaje tiene los *slots* vacíos pero, conforme el juego evoluciona, dichos *slots* se pueden rellenar atendiendo a varios criterios independientes. Por ejemplo, si el jugador acumula una cantidad determinada de puntos, entonces obtendría un *slot*. Si el personaje principal vence a un enemigo con un alto nivel, entonces obtendría otro *slot*. Por otra parte, el jugador podría hacer uso de estas habilidades especiales cuando así lo considere, siempre y cuando tenga al menos un *slot* relleno.

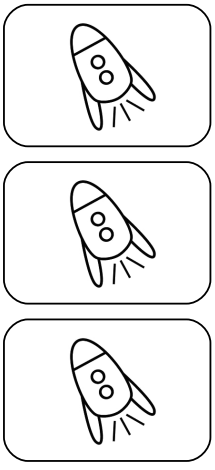


Figura 7.18: En los juegos arcade de aviones, los poderes especiales se suelen representar con cohetes para denotar la potencia de los mismos.

Este supuesto plantea la problemática de sincronizar el acceso concurrente a dichos *slots*, tanto para su consumo como para su generación. Además, hay que tener en cuenta que sólo será posible consumir un *slot* cuando haya al menos uno disponible. Es decir, la problemática discutida también plantea ciertas restricciones o condiciones que se han de satisfacer para que el jugador pueda lanzar una habilidad especial.

En este contexto, el uso de los monitores proporciona gran **flexibilidad** para modelar una solución a este supuesto. El siguiente listado de código muestra una posible implementación de la estructura de datos que podría dar soporte a la solución planteada, haciendo uso de los monitores de la biblioteca de hilos de ICE.

Como se puede apreciar, la clase definida es un tipo particular de monitor sin semántica recursiva, es decir, definido a partir de *IceUtil::Mutex*. Dicha clase tiene como variable miembro una cola de doble entrada que maneja tipos de datos genéricos, ya que la clase definida hace uso de una plantilla. Además, esta clase proporciona las dos operaciones típicas de *put()* y *get()* para añadir y obtener elementos. Hay, sin embargo, dos características importantes a destacar en el **diseño** de esta estructura de datos:

1. El acceso concurrente a la variable miembro de la clase se controla mediante el propio monitor, haciendo uso de la función *lock()* (líneas 11 y 17).

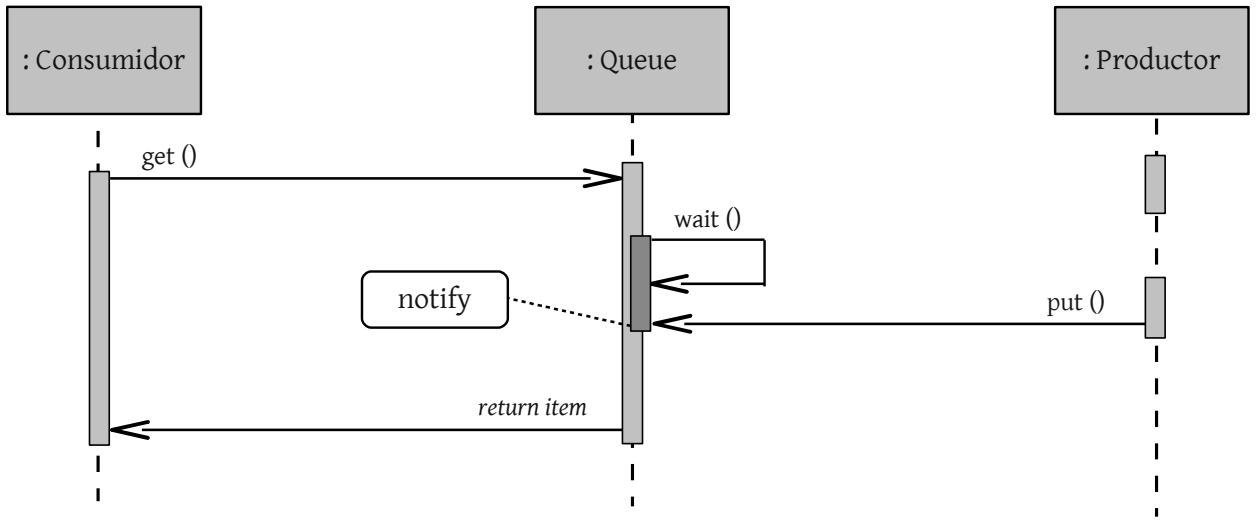


Figura 7.19: Representación gráfica del uso de un monitor.

Listado 7.28: Utilizando monitores

```
1  #include <IceUtil/Monitor.h>
2  #include <IceUtil/Mutex.h>
3  #include <deque>
4
5  using namespace std;
6
7  template<class T>
8  class Queue : public IceUtil::Monitor<IceUtil::Mutex> {
9  public:
10   void put (const T& item) { // Añade un nuevo item.
11     IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
12     _queue.push_back(item);
13     notify();
14   }
15
16   T get () { // Consume un item.
17     IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
18     while (_queue.size() == 0)
19       wait();
20     T item = _queue.front();
21     _queue.pop_front();
22     return item;
23   }
24
25 private:
26   deque<T> _queue; // Cola de doble entrada.
27 };
```

2. Si la estructura no contiene elementos, entonces el hilo se suspende mediante *wait()* (líneas 18-19) hasta que otro hilo que ejecute *put()* realice la invocación sobre *notify()* (línea 13).

Recuerde que para que un hilo bloqueado por *wait()* reanude su ejecución, otro hilo ha de ejecutar *notify()* y liberar el monitor mediante *unlock()*. Sin embargo, en el anterior listado de código no existe ninguna llamada explícita a *unlock()*. ¿Es incorrecta la solución? La respuesta es no, ya que la liberación del monitor se delega en *Lock* cuando la función *put()* finaliza, es decir, justo después de ejecutar la operación *notify()* en este caso particular.

No olvides...

Usar *Lock* y *TryLock* para evitar posibles interbloqueos causados por la generación de alguna excepción o una terminación de la función no prevista inicialmente.

Volviendo al ejemplo anterior, considere dos hilos distintos que interactúan con la estructura creada, de manera genérica, para almacenar los *slots* que permitirán la activación de habilidades especiales por parte del jugador virtual. Por ejemplo, el **hilo asociado al productor** podría implementarse como se muestra en el siguiente listado.

Listado 7.29: Hilo productor de slots

```
1  class Productor : public IceUtil::Thread {
2  public:
3    Productor (Queue<string> *_q):
4      _queue(_q) {}
5
6    void run () {
7      for (int i = 0; i < 5; i++) {
8        IceUtil::ThreadControl::sleep
9      (IceUtil::Time::seconds(rand() % 7));
10       _queue->put("TestSlot");
11     }
12   }
13
14 private:
15   Queue<string> *_queue;
16 };
```

Suponiendo que el código del consumidor sigue la misma estructura, pero extrayendo elementos de la estructura de datos compartida, entonces sería posible lanzar distintos hilos para comprobar que el acceso concurrente sobre los distintos *slots* se realiza de manera adecuada. Además, sería sencillo visualizar, mediante mensajes por la salida estándar, que efectivamente los hilos consumidores se suspenden en *wait()* hasta que hay al menos algún elemento en la estructura de datos compartida con los productores.

Antes de pasar a discutir en la sección 7.9 un caso de estudio para llevar a cabo el procesamiento de alguna tarea en segundo plano, resulta importante volver a discutir la solución planteada inicialmente para el manejo de monitores. En particular, la implementación de las funciones miembro *put()* y *get()* puede generar **sobrecarga** debido a que, cada vez que se añade un nuevo elemento a la estructura de datos, se realiza una invocación. Si no existen hilos esperando, la notificación se pierde. Aunque este hecho no conlleva ningún efecto no deseado, puede generar una reducción del rendimiento si el número de notificaciones se dispara.

Información adicional

Una vez el más, el uso de alguna estructura de datos adicional puede facilitar el diseño de una solución. Este tipo de planteamientos puede incrementar la eficiencia de la solución planteada aunque haya una mínima sobrecarga debido al uso y procesamiento de datos extra.

Una posible solución a este problema consiste en llevar un **control explícito** de la existencia de consumidores de información, es decir, de hilos que invoquen a la función *get()*. Para ello, se puede incluir una variable miembro en la clase *Queue*, planteando un esquema mucho más eficiente.

Como se puede apreciar en el siguiente listado de código, el hilo productor sólo llevará a cabo una notificación en el caso de que haya algún hilo consumidor en espera. Para ello, consulta el valor de la variable miembro *_consumidoresEsperando*.

Por otra parte, los hilos consumidores, es decir, los que invoquen a la función *get()* incrementan dicha variable antes de realizar un *wait()*, decrementándola cuando se despierten.

Note que el acceso a la variable miembro *_consumidoresEsperando* es exclusivo y se garantiza gracias a la adquisición del monitor justo al ejecutar la operación de generación o consumo de información por parte de algún hilo.

Listado 7.30: Utilizando monitores (II)

```
1  #include <IceUtil/Monitor.h>
2  #include <IceUtil/Mutex.h>
3  #include <deque>
4  using namespace std;
5
6  template<class T>
7  class Queue : public IceUtil::Monitor<IceUtil::Mutex> {
8  public:
9    Queue () : _consumidoresEsperando(0) {}
10
11   void put (const T& item) { // Añade un nuevo item.
12     IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
13     _queue.push_back(item);
14     if (_consumidoresEsperando) notify();
15   }
16
17   T get () { // Consume un item.
18     IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
19     while (_queue.size() == 0) {
20       try {
21     _consumidoresEsperando++;
22     wait();
23     _consumidoresEsperando--;
24       }
25       catch (...) {
26     _consumidoresEsperando--;
27     throw;
28       }
29     }
30     T item = _queue.front();
31     _queue.pop_front();
32     return item;
33   }
34
35 private:
36   deque<T> _queue; // Cola de doble entrada.
37   int _consumidoresEsperando;
38 };
```

7.7. Concurrencia en C++11

Una de las principales mejoras aportadas por C++11 es el soporte a la programación concurrente. De hecho, la creación de hilos resulta trivial, gracias a la **clase *Thread***[8], y sigue la misma filosofía que en otras bibliotecas más tradicionales (como *pthread*):

Para poder compilar este programa con la última versión del estándar puede utilizar el siguiente comando:

```
$ g++ -std=c++11 Thread_c++11.cpp -o Thread -pthread
```

[8] http://es.cppreference.com/w/cpp/thread

Listado 7.31: Uso básico de la clase Thread

```
#include <iostream>
#include <thread>
using namespace std;

void func (int x) {
    cout << "Dentro del hilo " << x << endl;
}

int main() {
    thread th(&func, 100);
    th.join();
    cout << "Fuera del hilo" << endl;
    return 0;
}
```

El **concepto de mutex**[9] como mecanismo básico de exclusión mutua también está contemplado por el lenguaje y permite acotar, de una manera sencilla y directa, aquellas partes del código que solamente deben ejecutarse por un hilo en un instante de tiempo (sección crítica). Por ejemplo, el siguiente fragmento de código muestra cómo utilizar un mutex para controlar el acceso exclusivo a una variable compartida por varios hilos:

Listado 7.32: Uso básico de la clase Mutex

```
int contador = 0;
mutex contador_mutex;

void anyadir_doble (int x) {
    int tmp = 2 * x;
    contador_mutex.lock();
    contador += tmp;
    contador_mutex.unlock();
}
```

C++11 ofrece incluso mecanismos que facilitan y simplifican el código de manera sustancial. Por ejemplo, la funcionalidad asociada al listado anterior se puede obtener, de manera alternativa, mediante el contenedor *atomic*:

Listado 7.33: Utilizando el contenedor atomic

```
#include <atomic>

atomic<int> contador(0);

void anyadir_doble (int x) {
    contador += 2 * x;
}
```

[9] http://es.cppreference.com/w/cpp/thread/mutex

7.7.1. Filósofos comensales en C++11

En esta sección se discutirá una posible implementación del problema de los filósofos comensales utilizando la funcionalidad proporcionada, desde el punto de vista de la programación concurrente, de C++11. Así mismo, también se introducirá el uso de algunas de las mejoras que incluye esta nueva versión del estándar, las cuales se discutirán con mayor profundidad en el módulo 3, *Técnicas Avanzadas de Desarrollo*.

El siguiente listado de código muestra la implementación de la **clase Palillo**. A diferencia del esquema planteado en la sección 7.6.3, esta implementación se basa en la composición en lugar de herencia con el objetivo de discutir las diferencias existentes entre ambos enfoques a la hora de manipular mutex. En principio, la primera opción a considerar debería ser la composición, es decir, un esquema basado en incluir una variable miembro de tipo mutex dentro de la declaración de la clase asociada al recurso compartido. Recuerde que las relaciones de herencia implican fuertes restricciones desde el punto de vista del diseño y se deberían establecer cuidadosamente.

Listado 7.34: Filósofos comensales en C++11 (Palillo)

```
1 class Palillo {
2 public:
3   mutex _mutex;
4 };
```

A continuación se muestra la función *comer()*, la cual incluye la funcionalidad asociada a coger los palillos. Note cómo esta función lambda o función anónima se define dentro del propio código de la función *main* (en línea) y propicia la generación de un código más claro y directo. Además, el uso de *auto* oculta la complejidad asociada al manejo de punteros a funciones.

Esta función recibe como argumentos (líneas 4-5) apuntadores a dos palillos, junto a sus identificadores, y el identificador del filósofo que va a intentar cogerlos para comer. En primer lugar, es interesante resaltar el uso de **std::lock** sobre los propios palillos en la línea 7 para evitar el problema clásico del interbloqueo de los filósofos comensales. Esta función hace uso de un algoritmo de evasión del interbloqueo y permite adquirir dos o más mutex mediante una única instrucción. Así mismo, también se utiliza el wrapper *lock_guard* (líneas 12 y 17) con el objetivo de indicar de manera explícita que los mutex han sido adquiridos y que éstos deben adoptar el propietario de dicha adquisición.

El resto del código de esta función muestra por la salida estándar mediante *std::cout* mensajes que indican el filósofo ha adquirido los palillos.

Una vez definida la funcionalidad general asociada a un filósofo, el siguiente paso consiste en instanciar los palillos a los propios filósofos.

Listado 7.35: Filósofos comensales en C++11 (Función comer)

```
1  int main () {
2    /* Función lambda (función anónima) */
3    /* Adiós a los punteros a funciones con auto */
4    auto comer = [](Palillo* pIzquierdo, Palillo* pDerecho,
5            int id_filosofo, int id_pIzquierdo, int id_pDerecho) {
6      /* Para evitar interbloqueos */
7      lock(pIzquierdo->_mutex, pDerecho->_mutex);
8
9      /* Wrapper para adquirir el mutex en un bloque de código */
10     /* Indica que el mutex ha sido adquirido y que debe
11      adoptar al propietario del cierre */
12     lock_guard<mutex> izquierdo(pIzquierdo->_mutex, adopt_lock);
13     string si = "\tFilósofo " + to_string(id_filosofo) +
```

```
14            " cogió el palillo " + to_string(id_pIzquierdo) + ".\n";
15     cout << si.c_str();
16
17     lock_guard<mutex> derecho(pDerecho->_mutex, adopt_lock);

18     string sd = "\tFilósofo " + to_string(id_filosofo) +
19            " cogió el palillo " + to_string(id_pDerecho) + ".\n";
20     cout << sd.c_str();
21
22     string pe = "Filósofo " + to_string(id_filosofo) + " come.\n";
23     cout << pe;
24
25     std::chrono::milliseconds espera(1250);
26     std::this_thread::sleep_for(espera);
27
28     /* Los mutex se desbloquean al salir de la función */
29   };
```

El listado que se muestra a continuación lleva a cabo la **instanciación de los palillos** (ver línea 9). Éstos se almacenan en un contenedor *vector* que inicialmente tiene una capacidad igual al número de filósofos (línea 1). Resulta interesante destacar el uso de *unique_ptr* en la línea 4 para que el programador se *olvide* de la liberación de los punteros asociados, la cual tendrá lugar, gracias a este enfoque, cuando queden fuera del ámbito de declaración. También se utiliza la función *std::move* en la línea 12, novedad en C++11, para copiar el puntero a *p1* en el vector *palillos*, anulándolo tras su uso.

Listado 7.36: Filósofos comensales en C++11 (Creación palillos)

```
1  static const int numero_filosofos = 5;
2
3  /* Vector de palillos */
4  vector< unique_ptr<Palillo> > palillos(numero_filosofos);
5
6  for (int i = 0; i < numero_filosofos; i++) {
7    /* El compilador infiere el tipo de la variable :-) */
8    /* unique_ptr para destrucción de objetos fuera del scope */
9    auto p1 = unique_ptr<Palillo>(new Palillo());
10   /* move copia c1 en la posición adecuada de palillos y
11      lo anula para usos posteriores */
12   palillos[i] = move(p1);
13  }
```

Finalmente, el código restante que se expone a continuación es el responsable de la **instanciación de los filósofos**. Éstos se almacenan en un vector que se instancia en la línea 2, el cual se rellena mediante un bucle for a partir de la línea 20. Sin embargo, antes se inserta el filósofo que tiene a su derecha el primer palillo y a su izquierda el último (líneas 6-17).

Note cómo en cada iteración de este bucle se crea un nuevo hilo, instanciando la clase *thread* (línea 21). Cada instancia de esta clase maneja un apuntador a la función *comer*, la cual recibe como argumentos los apuntadores a los palillos que correspondan a cada filósofo y los identificadores correspondientes para mostrar por la salida estándar los mensajes reflejan la evolución de la simulación.

La última parte del código, en las líneas 32-34, lleva a cabo la ejecución efectiva de los hilos y la espera a la finalización de los mismos mediante la función *thread::join()*.

Listado 7.37: Filósofos comensales en C++11 (Creación filósofos)

```
1  /* Vector de filósofos */
```

```
vector<thread> filosofos(numero_filosofos);

/* Primer filósofo */
/* A su derecha el palillo 1 y a su izquierda el palillo 5 */
filosofos[0] = thread(comer,
               /* Palillo derecho (1) */
               palillos[0].get(),
               /* Palillo izquierdo (5) */
               palillos[numero_filosofos - 1].get(),
               /* Id filósofo */
               1,
               /* Id palillo derecho */
               1,
               /* Id palillo izquierdo */
               numero_filosofos
               );

/* Restos de filósofos */
for (int i = 1; i < numero_filosofos; i++) {
  filosofos[i] = (thread(comer,
              palillos[i - 1].get(),
              palillos[i].get(),
              i + 1,
              i,
              i + 1
              )
          );
 }

/* A comer... */
for_each(filosofos.begin(),
     filosofos.end(),
     mem_fn(&thread::join));
```

7.8. *Multi-threading* en Ogre3D

Desde un punto de vista general, la posible integración de hilos de ejecución adicionales con respecto al núcleo de ejecución de Ogre se suele plantear con aspectos diferentes al meramente gráfico. De hecho, es poco común que el proceso de *rendering* se delegue en un segundo plano debido a la propia naturaleza asíncrona de la GPU (Graphic Processing Unit). Por lo tanto, no existe una necesidad real de utilizar algún esquema de sincronización para evitar condiciones de carrera o asegurar la exclusión mutua en determinadas partes del código.

Sin embargo, sí que es posible delegar en un **segundo plano** aspectos tradicionalmente independientes de la parte gráfica, como por ejemplo el módulo de Inteligencia Artificial o la carga de recursos. En esta sección, se discutirán las características básicas que Ogre3D ofrece para la carga de recursos en segundo plano.

Figura 7.20: Ogre proporciona mecanismos para llevar a cabo la carga de recursos en segundo plano.

En primer lugar, puede ser necesario compilar Ogre con soporte para *multi-threading*, típicamente soportado por la biblioteca *boost*. Para ejemplificar este problema, a continuación se detallan las instrucciones para compilar el código fuente de *Ogre 1.8.1*[10].

Antes de nada, es necesario asegurarse de que la biblioteca de hilos de *boost* está instalada y es recomendable utilizar *cmake* para generar el *Makefile* que se utilizará para compilar el código fuente. Para ello, en sistemas operativos Debian GNU/Linux y derivados, hay que ejecutar los siguientes comandos:

```
$ sudo apt-get install libboost-thread-dev
$ sudo apt-get install cmake cmake-qt-gui
```

[10] http://www.ogre3d.org/download/source

A continuación, es necesario descomprimir el código de Ogre y crear un directorio en el que se generará el resultado de la compilación:

```
$ tar xjf ogre_src_v1-8-1.tar.bz2
$ mkdir ogre_build_v1-8-1
```

El siguiente paso consiste en ejecutar la interfaz gráfica que nos ofrece *cmake* para hacer explícito el soporte multi-hilo de Ogre. Para ello, simplemente hay que ejecutar el comando *cmake-gui* y, a continuación, especificar la ruta en la que se encuentra el código fuente de Ogre y la ruta en la que se generará el resultado de la compilación, como se muestra en la figura 7.21. Después, hay que realizar las siguientes acciones:

1. Hacer *click* en el botón *Configure*.
2. Ajustar el parámetro *OGRE_CONFIG_THREADS*[11].
3. Hacer *click* de nuevo en el botón *Configure*.
4. Hacer *click* en el botón *Generate* para generar el archivo *Makefile*.

Es importante recalcar que el **parámetro *OGRE_CONFIG_THREADS*** establece el tipo de soporte de Ogre respecto al modelo de hilos. Un valor de 0 desabilita el soporte de hilos. Un valor de 1 habilita la carga completa de recursos en segundo plano. Un valor de 2 activa solamente la preparación de recursos en segundo plano[12].

Finalmente, tan sólo es necesario compilar y esperar a la generación de todos los ejecutables y las bibliotecas de Ogre. Note que es posible optimizar este proceso indicando el número de procesadores físicos de la máquina mediante la opción $-j$ de *make*.

```
$ cd ogre_build_v1-8-1
$ make -j 2
```

Para ejemplificar la carga de recursos en segundo plano, se retomará el ejemplo discutido en la sección 6.2.2. En concreto, el siguiente listado de código muestra cómo se ha modificado la carga de un *track* de música para explicitar que se haga en segundo plano.

Como se puede apreciar, en la línea (13) se hace uso de la función *setBackgroundLoaded()* con el objetivo de indicar que el recurso se cargará en segundo plano. A continuación, en la siguiente línea, se añade un objeto de la clase *MyListener*, la cual deriva a su vez de la clase ***Ogre::Resource::Listener*** para controlar la notificación de la carga del recurso en segundo plano.

Callbacks

El uso de funciones de retrollamada permite especificar el código que se ejecutará cuando haya finalizado la carga de un recurso en segundo plano. A partir de entonces, será posible utilizarlo de manera normal.

En este ejemplo se ha sobreescrito la función *backgroundLoadingComplete()* que permite conocer cuándo se ha completado la carga en segundo plano del recurso. La documentación relativa a la clase *Ogre::Resource::Listener*[13] discute el uso de otras funciones de retrollamada que resultan útiles para cargar contenido en segundo plano.

Listado 7.38: Carga en segundo plano de un recurso en Ogre

```
1 TrackPtr
2 TrackManager::load
```

[11] http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Building+Ogre+With+CMake

[12] Según los desarrolladores de Ogre, sólo se recomienda utilizar un valor de 0 o de 2 en sistemas GNU/-Linux

[13] `http://www.ogre3d.org/docs/api/html/classOgre_1_1Resource_1_1Listener.html`

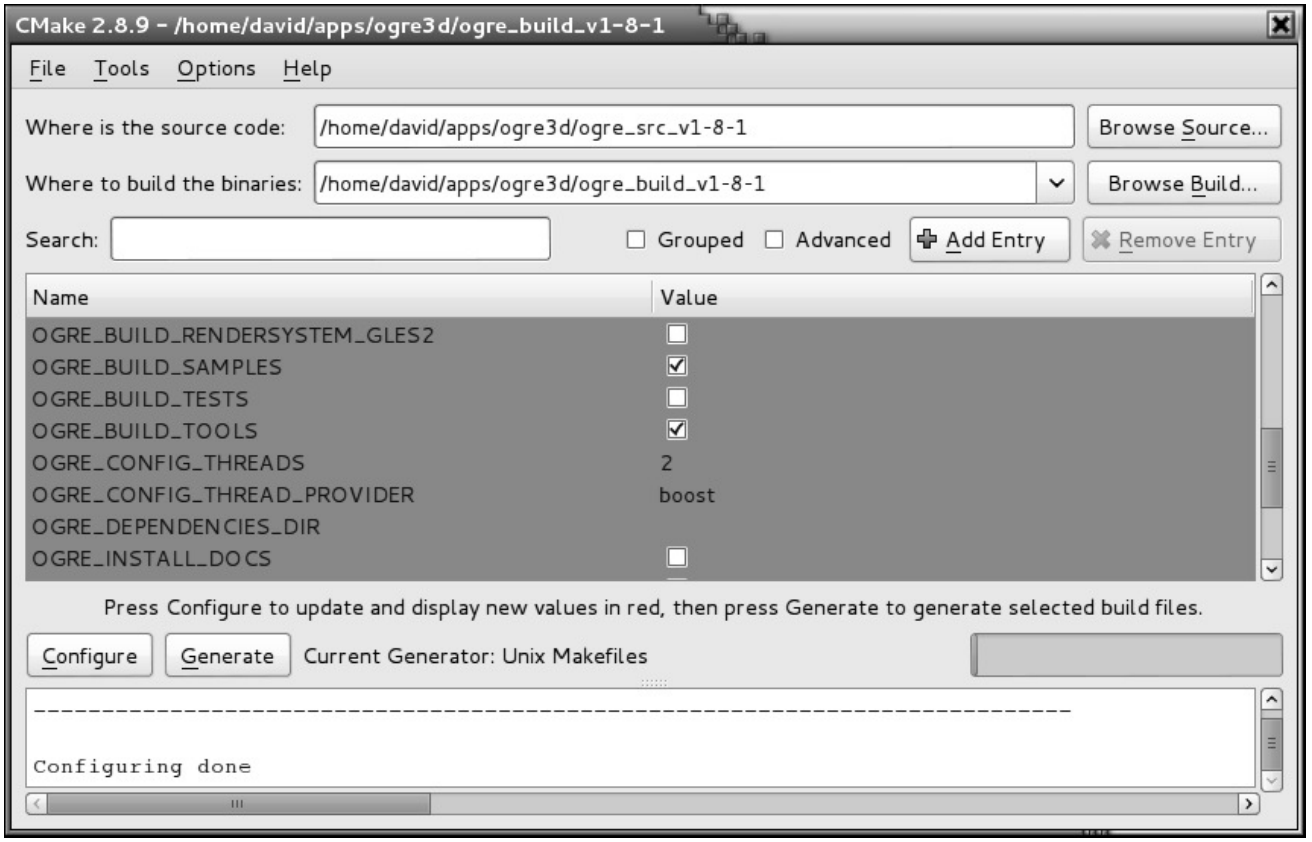


Figura 7.21: Generando el fichero *Makefile* mediante *cmake* para compilar Ogre 1.8.1.

```
 3 (const Ogre::String& name, const Ogre::String& group)
 4 {
 5   // Obtención del recurso por nombre...
 6   TrackPtr trackPtr = getByName(name);
 7
 8   // Si no ha sido creado, se crea.
 9   if (trackPtr.isNull())
10     trackPtr = create(name, group);
11
12   // Carga en segundo plano y listener.
13   trackPtr->setBackgroundLoaded(true);
14   trackPtr->addListener(new MyListener);
15
16   // Escala la carga del recurso en segundo plano.
17   if (trackPtr->isBackgroundLoaded())
18     trackPtr->escalateLoading();
19   else
20     trackPtr->load();
21
22   return trackPtr;
23 }
```

Otra posible variación en el diseño podría haber consistido en delegar la notificación de la función de retrollamada a la propia clase *TrackPtr*, es decir, que esta clase heredase de *Ogre::Resource::Listener*.

Listado 7.39: Clase MyListener

```
1  #ifndef __MYLISTENERH__
2  #define __MYLISTENERH__
3
4  #include <OGRE/OgreResource.h>
5  #include <iostream>
6
7  using namespace std;
8
9  class MyListener : public Ogre::Resource::Listener {
10
11   void backgroundLoadingComplete (Ogre::Resource* r) {
12     cout << "Carga en segundo plano completada..." << endl;
13   }
14
15 };
16
17 #endif
```

7.9. Caso de estudio. Procesamiento en segundo plano mediante hilos

Figura 7.22: Aspecto gráfico de un juego tipo *Simon*.

En esta sección se discute cómo llevar a cabo un procesamiento adicional al hilo de control manejado por Ogre. Para ello, se hace uso de las herramientas proporcionadas por la biblioteca de hilos de ZeroC ICE, estudiadas en las secciones anteriores.

En concreto, el problema que se plantea consiste en diseñar e implementar un sencillo **módulo de Inteligencia Artificial** para un juego de tipo *Simon*[14], el cual será el responsable de la generación de las distintas secuencias de juego. Más allá de la complejidad del módulo de IA, resulta más interesante en este punto afrontar el problema de la creación y gestión de hilos de control adicionales. En el módulo 4, *Desarrollo de Componentes*, se discuten diversas técnicas de IA que se pueden aplicar a la hora de implementar un juego.

Antes de pasar a discutir la implementación planteada, recuerde que en el juego *Simon* el jugador ha de ser capaz de memorizar y repetir una secuencia de colores generada por el dispositivo de juego. En este contexto, el módulo de IA sería el encargado de ir generando la secuencia a repetir por el jugador.

El siguiente listado de código muestra la declaración del hilo dedicado al módulo de IA. Como se puede apreciar, la clase *AIThread* hereda de *IceUtil::Thread*. Las variables miembro de esta clase son las siguientes:

Updating threads...

Idealmente, el módulo de IA debería aprovechar los recursos disponibles, no utilizados por el motor de *rendering*, para actualizar su estado. Además, se debería contemplar la posibilidad de sincronización entre los mismos.

- **_delay**, que mantiene un valor temporal utilizado para *dormir* al módulo de IA entre actualizaciones.
- **_mutex**, que permite controlar el acceso concurrente al estado de la clase.
- **_seq**, que representa la secuencia de colores, mediante valores enteros, generada por el módulo de IA.

[14] http://code.google.com/p/videojuegos-2011-12

En la solución planteada se ha optado por utilizar el mecanismo de exclusión mutua proporcionado por ICE para sincronizar el acceso concurrente al estado de *AIThread* con un doble propósito: i) ilustrar de nuevo su utilización en otro ejemplo y ii) recordar que es posible que distintos hilos de control interactúen con el módulo de IA.

Note cómo en la línea (23) se ha definido un manejador para las instancias de la clase *AIThread* utilizando *IceUtil::Handle* con el objetivo de gestionar de manera inteligente dichas instancias. Recuerde que la función miembro *run()* especificará el comportamiento del hilo y es necesario implementarla debido al contracto funcional heredado de la clase *IceUtil::Thread*.

Listado 7.40: Clase AIThread

```
1  #ifndef __AI__
2  #define __AI__
3
4  #include <IceUtil/IceUtil.h>
5  #include <vector>
6
7  class AIThread : public IceUtil::Thread {
8
9   public:
10   AIThread (const IceUtil::Time& delay);
11   int getColorAt (const int& index) const;
12   void reset ();
13   void update ();
14
15   virtual void run ();
16
17  private:
18   IceUtil::Time _delay;  // Tiempo entre actualizaciones.
19   IceUtil::Mutex _mutex; // Cerrojo para acceso exclusivo.
20   std::vector<int> _seq; // Secuencia de colores.
21 };
22
23 typedef IceUtil::Handle<AIThread> AIThreadPtr; // Smart pointer.
24
25 #endif
```

Actualmente, las **responsabilidades** del hilo encargado de procesar la IA del juego *Simon* son dos. Por una parte, actualizar periódicamente la secuencia de colores a repetir por el usuario. Esta actualización se llevaría a cabo en la función *run()*, pudiendo ser más o menos sofisticada en función del nivel de IA a implementar. Por otra parte, facilitar la obtención del siguiente color de la secuencia cuando el jugador haya completado la subsecuencia anterior.

El siguiente listado de código muestra una posible implementación de dichas funciones. Básicamente, el hilo de IA genera los colores de manera aleatoria, pero sería posible realizar una implementación más sofisticada para, por ejemplo, modificar o invertir de algún modo la secuencia de colores que se va generando con el paso del tiempo.

Considere que el *delay* de tiempo existente entre actualización y actualización del módulo de IA debería estar condicionado por la actual tasa de *frames* por segundo del juego. En otras palabras, la tasa de actualización del módulo de IA será más o menos exigente en función de los recursos disponibles con el objetivo de no penalizar el rendimiento global del juego.

Uso de hilos

Considere el uso de hilos cuando realmente vaya a mejorar el rendimiento de su aplicación. Si desde el hilo de control principal se puede atender la lógica de IA, entonces no sería necesario delegarla en hilos adicionales.

También es muy importante considerar la **naturaleza del juego**, es decir, las necesidades reales de actualizar el módulo de IA. En el caso del juego *Simon*, está necesidad no sería especialmente relevante considerando el intervalo de tiempo existente entre la generación de una secuencia y la generación del siguiente elemento que extenderá la misma. Sin embargo, en juegos en los que intervienen un gran número de *bots* con una gran interactividad, la actualización del módulo de IA se debería producir con mayor frecuencia.

Listado 7.41: AIThread::run() y AIThread::update()

```
1  int AIThread::getColorAt (const int& index) const {
2    IceUtil::Mutex::Lock lock(_mutex);
3    return _seq[index];
4  }
5
6  void AIThread::update () {
7    IceUtil::Mutex::Lock lock(_mutex);
8    // Cálculos complejos del módulo de IA.
9    _seq.push_back(rand() % 4);
10 }
11
12 void AIThread::run () {
13   while (true) {
14     // Calcular nueva secuencia...
15     std::cout << "Updating..." << std::endl;
16     update();
17     IceUtil::ThreadControl::sleep(_delay);
18   }
19 }
```

La siguiente figura muestra el diagrama de interacción que refleja la gestión del hilo de IA desde la clase principal *MyApp* (en este ejemplo no se considera el esquema discutido en la sección 6.1.4 relativo a los estados de juego con Ogre). Como se puede apreciar, desde dicha clase principal se crea la instancia de *AIThread* y se ejecuta la función *start()*. A partir de ahí, el hilo continuará su ejecución de acuerdo al comportamiento definido en su función *run()*.

Rendimiento con hilos

El uso de hilos implica cambios de contexto y otras operaciones a nivel de sistema operativo que consumen miles de ciclos de ejecución. Evalúe siempre el impacto de usar una solución multi-hilo.

Típicamente, el módulo de IA actualizará parte del estado del juego en cuestión, en función de las características de éste. Por ejemplo, en un juego de acción, el comportamiento internos de los personajes, implementado en el módulo de IA, gobernará parte de las acciones de los mismos, como por ejemplo la disposición para avanzar físicamente hacia un objetivo. Evidentemente, este tipo de acciones implican una **interacción** con el motor gráfico.

El siguiente listado de código muestra la parte de la clase *MyApp* desde la que se crea el hilo responsable del procesamiento de la IA.

Note cómo en este ejemplo se utiliza la función *detach()* para desligar el hilo de la IA respecto del hilo de control principal. Este planteamiento implica que, para evitar comportamientos inesperados, el desarrollador ha de asegurarse de que el hijo previamente *desligado* termine su ejecución antes de que el programa principal lo haga. En el código fuente desarrollado esta condición se garantiza al utilizar el manejador *IceUtil::Handle* para la clase *AIThread*.

Listado 7.42: MyApp::start(); creación de AIThread

```
1  int MyApp::start() {
2    _root = new Ogre::Root();
3    if(!_root->restoreConfig())
4    {
5      _root->showConfigDialog();
6      _root->saveConfig();
```

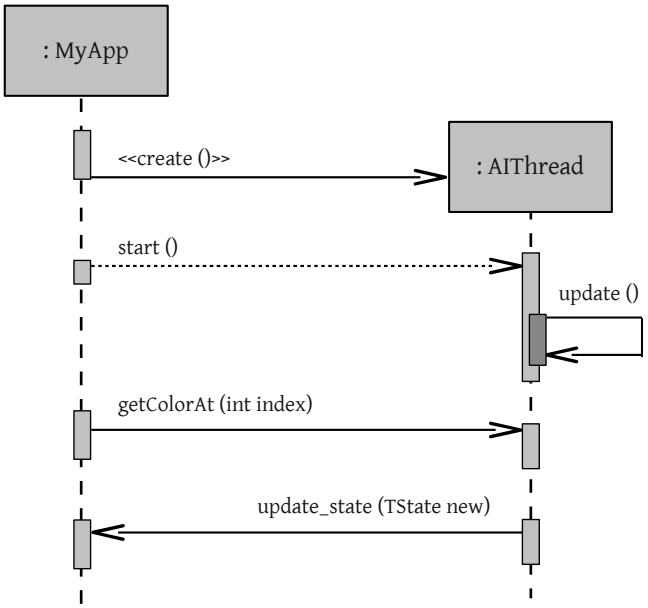
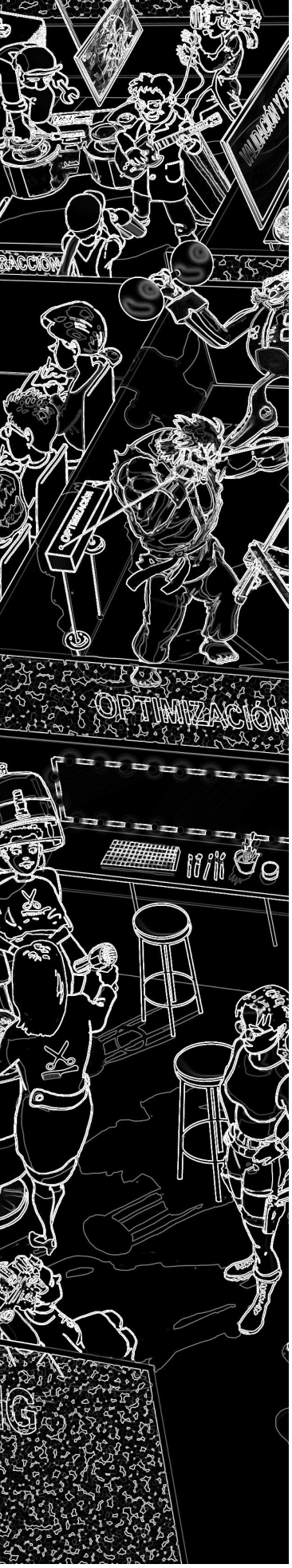


Figura 7.23: Diagrama de interacción entre la clase principal del juego y *AIThread*.

```
 7    }
 8
 9    _AI = new AIThread(IceUtil::Time::seconds(1));
10    IceUtil::ThreadControl tc = _AI->start();
11    // Se desliga del hilo principal.
12    tc.detach();
13
14    // ...
15  }
```

Módulo #2

Programación Gráfica



Programación Gráfica

El objetivo de este módulo titulado «*Programación Gráfica*» del Curso de Experto en Desarrollo de Videojuegos es cubrir los aspectos esenciales relativos al desarrollo de un motor gráfico interactivo. En este contexto, el presente módulo cubre aspectos esenciales y básicos relativos a los fundamentos del desarrollo de la parte gráfica, como por ejemplo el pipeline gráfico, como elemento fundamental de la arquitectura de un motor de juegos, las bases matemáticas, las APIs de programación gráfica, el uso de materiales y texturas, la iluminación o los sistemas de partículas. Así mismo, el presente módulo también discute aspectos relativos a la exportación e importación de datos, haciendo especial hincapié en los formatos existentes para tratar con información multimedia. Finalmente, se pone de manifiesto la importancia del uso de elementos directamente conectados con el motor gráfico, como la simulación física, con el objetivo de dotar de más realismo en el comportamiento de los objetos y actores que intervienen en el juego.

Capítulo 8

Fundamentos de Gráficos Tridimensionales

Carlos González Morcillo

Uno de los aspectos que más llaman la atención en los videojuegos actuales son sus impactantes gráficos 3D. En este primer capítulo introduciremos los conceptos básicos asociados al pipeline en gráficos 3D. Se estudiarán las diferentes etapas de transformación de la geometría hasta su despliegue final en coordenadas de pantalla. En la segunda parte del capítulo estudiaremos algunas de las capacidades básicas del motor gráfico de videojuegos y veremos los primeros ejemplos básicos de uso de Ogre.

8.1. Introducción

¡¡Aún más rápido!!

Si cada *frame* tarda en desplegarse más de 40ms, no conseguiremos el mínimo de 25 fps (*frames per second*) establecido como estándar en cine. En videojuegos, la frecuencia recomendable *mínima* es de unos 50 fps.

Desde el punto de vista del usuario, un videojuego puede definirse como una aplicación software que responde a una serie de eventos, *redibujando* la escena y generando una serie de respuestas adicionales (sonido en los altavoces, vibraciones en dispositivos de control, etc...).

El *redibujado* de esta escena debe realizarse lo más rápidamente posible. La capa de aplicación en el despliegue gráfico es una de las actividades que más ciclos de CPU consumen (incluso, como veremos en la sección 8.3, con el apoyo de las modernas GPUs *Graphics Processing Unit* existentes en el mercado).

Habitualmente el motor gráfico trabaja con geometría descrita mediante mallas triangulares. Las técnicas empleadas para optimizar el despliegue de esta geometría, junto con las propiedades de materiales, texturas e iluminación, varían dependiendo del tipo de videojuego que se está desarrollando. Por ejemplo, en un simulador de vuelo, el tratamiento que debe darse de la geometría distante (respecto de la posición de la cámara virtual) debe ser diferente que la empleada para optimizar la visualización de interiores en un videojuego de primera persona.

Uno de los errores habituales en videojuegos *amateur* es la falta de comunicación clara entre programadores y diseñadores. El desarrollador debe especificar claramente las capacidades soportadas por el motor gráfico al equipo de modeladores, animadores y equipo artístico en general. Esta información debe comprender tanto las técnicas de despliegue soportadas, como el número de polígonos disponibles para el personaje principal, enemigos, fondos, etc...

A un alto nivel de abstracción podemos ver el proceso de *render* como el encargado de convertir la descripción de una escena tridimensional en una imagen bidimensional. En los primeros años de estudio de esta disciplina, la investigación se centró en cómo resolver problemas relacionados con la detección de superficies visibles, sombreado básico, etc. Según se encontraban soluciones a estos problemas, se continuó el estudio de algoritmos más precisos que simularan el comportamiento de la luz de una forma más precisa.

En esencia, el proceso de *rendering* de una escena 3D requiere los siguientes elementos:

- **Superficies**. La geometría de los objetos que forman la escena debe ser definida empleando alguna representación matemática, para su posterior procesamiento por parte del ordenador.
- **Cámara**. La situación del visor debe ser definida mediante un par (posición, rotación) en el espacio 3D. El plano de imagen de esta cámara virtual definirá el resultado del proceso de *rendering*. Como se muestra en la Figura 8.1, para imágenes generadas en perspectiva, el volumen de visualización define una pirámide truncada que selecciona los objetos que serán representados en la escena. Esta pirámide se denomina *Frustum*.
- **Fuentes de luz**. Las fuentes de luz emiten rayos que interactúan con las superficies e impactarán en el plano de imagen. Dependiendo del modo de simulación de estos impactos de luz (de la resolución de la denominada *ecuación de render*), tendremos diferentes *métodos* de rendering.
- **Propiedades de las superficies**. En este apartado se incluyen las propiedades de materiales y texturas que describen el modelo de *rebote* de los fotones sobre las superficies.

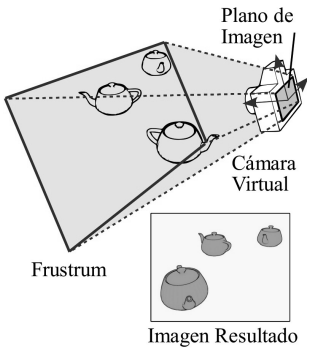


Figura 8.1: Descripción general del proceso de *rendering*.

Tiempo de cómputo

En síntesis de imagen realista (como por ejemplo, las técnicas de rendering empleadas en películas de animación) el cálculo de un solo fotograma de la animación puede requerir desde varias horas hasta días, empleando computadores de altas prestaciones.

Uno de los principales objetivos en síntesis de imagen es el realismo. En general, según el método empleado para la resolución de la *ecuación de render* tendremos diferentes niveles de realismo (y diferentes tiempos de cómputo asociados). El principal problema en gráficos en tiempo real es que las imágenes deben ser generadas muy rápidamente. Eso significa, como hemos visto anteriormente, que el motor gráfico dispone de menos de 40 ms para generar cada imagen. Habitualmente este tiempo es incluso menor, ya que es necesario reservar tiempo de CPU para otras tareas como el cálculo de la Inteligencia Artificial, simulación física, sonido...

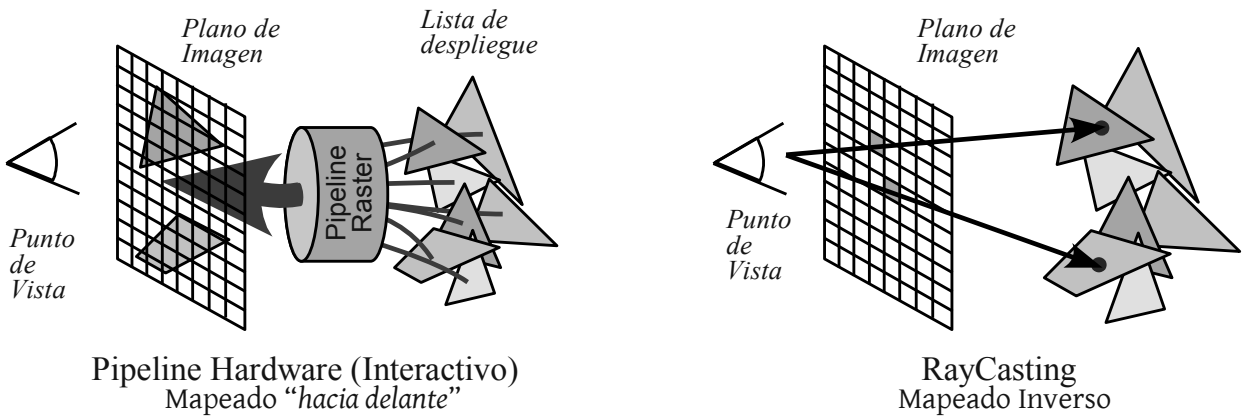


Figura 8.2: Diferencias entre las técnicas de despliegue interactivas (métodos basados en *ScanLine*) y realistas (métodos basados en RayCasting). En el Pipeline Hardware, la selección del píxel más cercano relativo a cada triángulo se realiza directamente en Hardware, empleando información de los *fragmentos* (ver Sección 8.2).

Los primeros métodos de sombreado de superficies propuestos por *Gouraud* y *Phong* no realizaban ninguna simulación física de la reflexión de la luz, calculando únicamente las contribuciones locales de iluminación. Estos modelos tienen en cuenta la posición de la luz, el observador y el vector normal de la superficie. Pese a su falta de realismo, la facilidad de su cómputo hace que estas aproximaciones sigan siendo ampliamente utilizadas en el desarrollo de videojuegos.

RayTracing

El algoritmo original del *RayCasting* de Appel, fue el precursor del método de *RayTracing* (*Trazado de Rayos*) de Whitted de 1980. El método de RayTracing sirvió de base para los principales métodos de síntesis de imagen hiperrealistas que se emplean en la actualidad (Metrópolis, Path Tracing, etc...).

En 1968 Arthur Appel describió el primer método para generar imágenes por computador lanzando rayos desde el punto de vista del observador. En este trabajo, generaba la imagen resultado en un plotter donde dibujaba punto a punto el resultado del proceso de render. La idea general del método de RayCasting es lanzar rayos desde el plano de imagen, uno por cada píxel, y encontrar el punto de intersección más cercano con los objetos de la escena. La principal ventaja de este método frente a los métodos de tipo *scanline* que emplean *zbuffer* es que es posible generar de forma consistente la imagen que represente el mundo 3D, ya que cualquier objeto que pueda ser descrito mediante una ecuación puede ser representado de forma correcta mediante RayCasting.

Como puede verse en la Figura 8.2, existen diferencias importantes entre el método de despliegue que implementan las tarjetas aceleradoras 3D (y en general los motores de visualización para aplicaciones interactivas) y el método de RayCasting. El *pipeline* gráfico de aplicaciones interactivas (como veremos en la sección 8.2) puede describirse de forma general como el que, a partir de una lista de objetos geométricos a representar y, tras aplicar la serie de transformaciones geométricas sobre los objetos, la vista y la perspectiva, obtienen una imagen raster dependiente del dispositivo de visualización. En este enfoque, las primitivas se ordenan según la posición de la cámara y sólo las visibles serán dibujadas. Por el contrario, en métodos de síntesis de imagen realista (como la aproximación inicial de RayCasting) calcula los rayos que pasan por cada píxel de la imagen y recorre la lista de objetos, calculando la intersección (si hay alguna) con el objeto más cercano. Una vez obtenido el punto de intersección, se evalúa (empleando un modelo de iluminación) el valor de sombreado correspondiente a ese píxel.

8.2. El Pipeline Gráfico

Para obtener una imagen de una escena 3D definida en el *Sistema de Referencia Universal*, necesitamos definir un sistema de referencia de coordenadas para los **parámetros de visualización** (también denominados *parámetros de cámara*). Este sistema de referencia nos definirá el plano de proyección, que sería el equivalente de la zona de la cámara sobre la que se registrará la imagen[1]. De este modo se transfieren los objetos al sistema de coordenadas de visualización y finalmente se proyectan sobre el plano de visualización (ver Figura 8.4).

El proceso de visualizar una escena en 3D mediante gráficos por computador es similar al que se realiza cuando se toma una fotografía real. En primer lugar hay que situar el *trípode* con la cámara en un lugar del espacio, eligiendo así una posición de visualización. A continuación, rotamos la cámara eligiendo si la fotografía la tomaremos en vertical o en apaisado, y apuntando al motivo que queremos fotografiar. Finalmente, cuando disparamos la fotografía, sólo una pequeña parte del mundo queda representado en la imagen 2D final (el resto de elementos son *recortados* y no aparecen en la imagen).

La Figura 8.3 muestra los pasos generales del *Pipeline* asociado a la transformación de una escena 3D hasta su representación final en el dispositivo de visualización (típicamente una pantalla con una determinada resolución).

El *Pipeline* está dividido en etapas funcionales. Al igual que ocurre en los *pipeline* de fabricación industrial, algunas de estas etapas se realizan en paralelo y otras secuencialmente. Idealmente, si dividimos un proceso en n etapas se incrementará la velocidad del proceso en ese factor n. Así, la velocidad de la cadena viene determinada por el tiempo requerido por la etapa más lenta.

Como señala Akenine-Möler [8], el pipeline interactivo se divide en tres etapas conceptuales de *Aplicación*, *Geometría* y *Rasterización* (ver Figura 8.3). A continuación estudiaremos estas etapas.

8.2.1. Etapa de Aplicación

La etapa de aplicación se ejecuta en la CPU. Actualmente la mayoría de las CPUs son multinúcleo, por lo que el diseño de esta aplicación se realiza mediante diferentes hilos de ejecución en paralelo. Habitualmente en esta etapa se ejecutan tareas asociadas al cálculo de la posición de los modelos 3D mediante simulaciones físicas, detección de colisiones, gestión de la entrada del usuario (teclado, ratón, joystick...). De igual modo, el uso de estructuras de datos de alto nivel para la aceleración del despliegue (reduciendo el número de polígonos que se envían a la GPU) se implementan en la etapa de aplicación.

8.2.2. Etapa de Geometría

En su tortuoso viaje hasta la pantalla, cada objeto 3D se transforma en diferentes sistemas de coordenadas. Originalmente, como se muestra en la Figura 8.4, un objeto tiene su propio *Sistema de Coordenadas Local* que nos definen las **Coordenadas de Modelo**, por lo que *desde su punto de vista* no está transformado.

[1] En el mundo físico, la *película* en antiguas cámaras analógicas, o el sensor de imagen de las cámaras digitales.

Figura 8.3: Pipeline general en gráficos 3D.

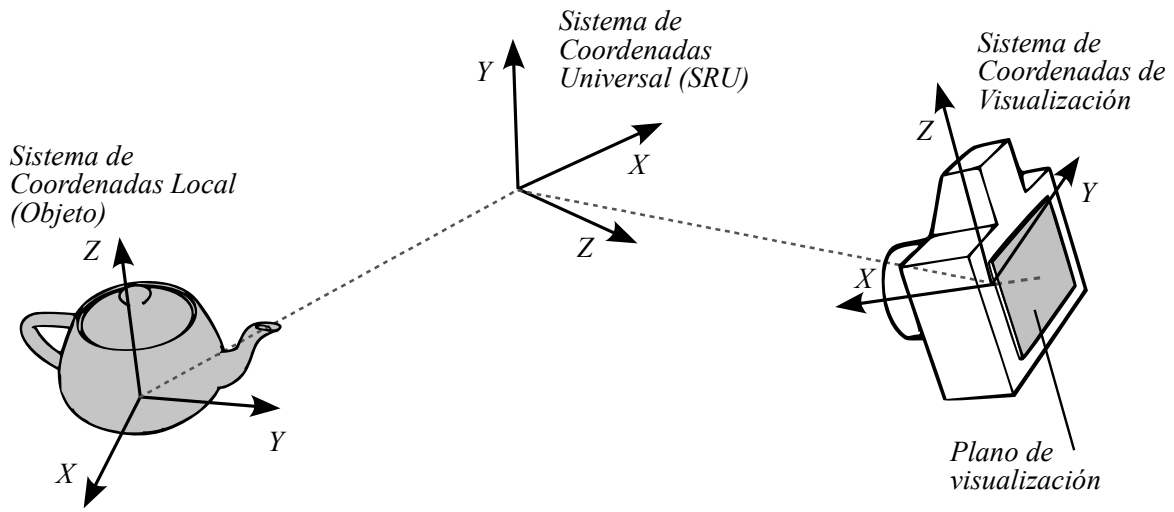


Figura 8.4: Sistema de coordenadas de visualización y su relación con otros sistemas de coordenadas de la escena.

A los vértices de cada modelo se le aplican la denominada **Transformación de Modelado** para posicionarlo y orientarlo respecto del *Sistema de Coordenadas Universal*, obteniendo así las denominadas **Coordenadas Universales** o *Coordenadas del Mundo*.

Instancias

Gracias a la separación entre *Coordenadas de Modelo* y *Transformación de Modelado* podemos tener diferentes instancias de un mismo modelo para construir una escena a las que aplicamos diferentes transformaciones. Por ejemplo, para construir un templo romano tendríamos un único objeto de columna y varias *instancias* de la columna a las que hemos aplicado diferentes traslaciones.

Como este sistema de coordenadas es único, tras aplicar la transformación de modelado a cada objeto, ahora todas las coordenadas estarán expresadas en el mismo espacio.

La posición y orientación de la cámara nos determinará qué objetos aparecerán en la imagen final. Esta cámara tendrá igualmente unas coordenadas universales. El propósito de la **Transformación de Visualización** es posicionar la cámara en el origen del *SRU*, apuntando en la dirección negativa del eje Z y el eje Y hacia arriba. Obtenemos de este modo las **Coordenadas de Visualización** o *Coordenadas en Espacio Cámara* (ver Figura 8.5).

Habitualmente el pipeline contiene una etapa adicional intermedia que se denomina **Vertex Shader** *Sombreado de Vértice* que consiste en obtener la representación del material del objeto modelando las transformaciones en las fuentes de luz, utilizando los vectores normales a los puntos de la superficie, información de color, etc. Es conveniente en muchas ocasiones transformar las posiciones de estos elementos (fuentes de luz, cámara, ...) a otro espacio (como *Coordenadas de Modelo*) para realizar los cálculos.

La **Transformación de Proyección** convierte el **volumen de visualización** en un cubo unitario (ver Sección 8.2.4). Este *volumen de visualización* se define mediante planos de recorte 3D y define todos los elementos que serán visualizados. En la figura 8.5 se representa mediante el volumen sombreado. Existen multitud de métodos de proyección, aunque como veremos más adelante, los más empleados son la ortográfica (o paralela) y la perspectiva.

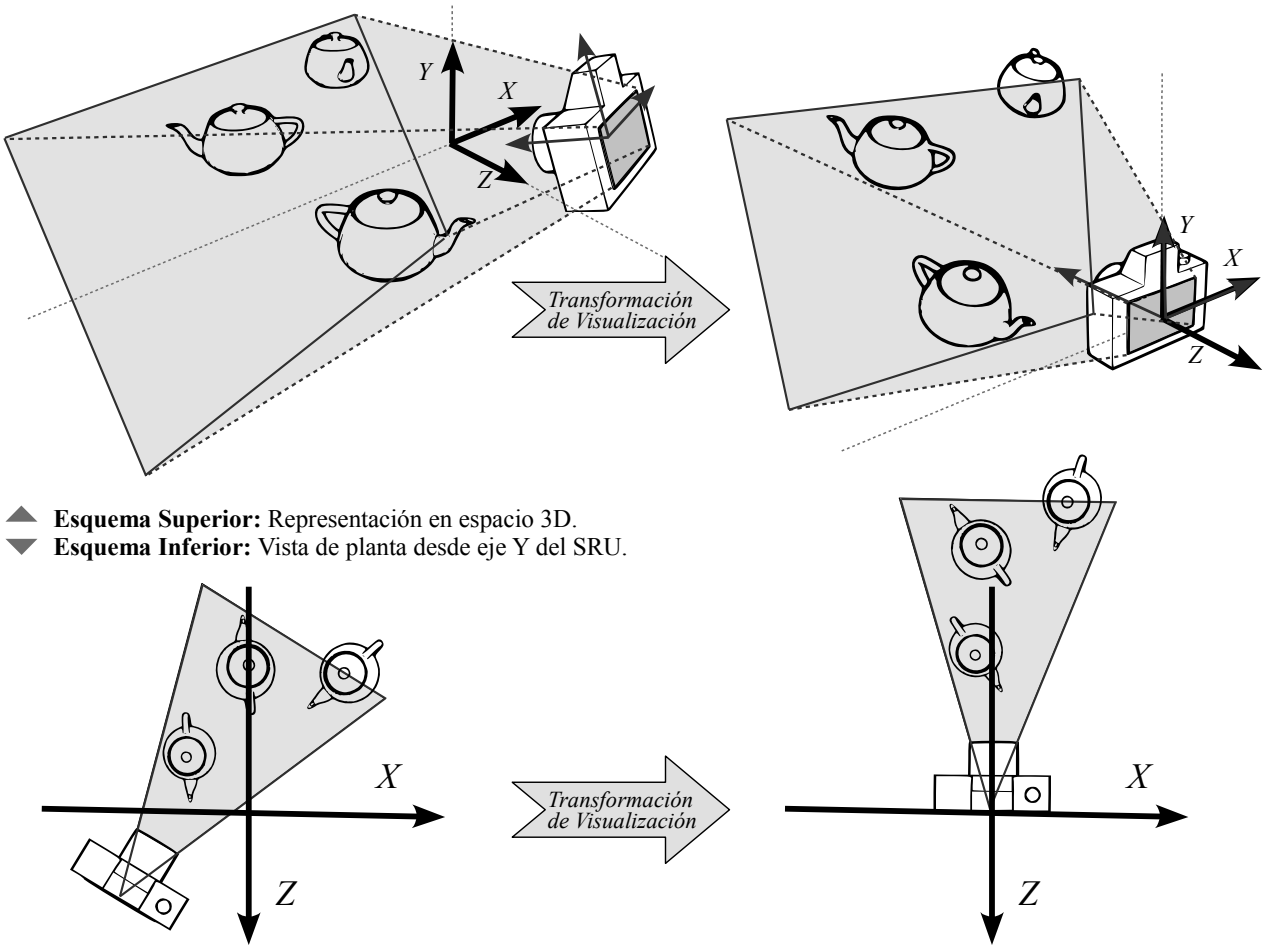


Figura 8.5: El usuario especifica la posición de la cámara (izquierda) que se transforma, junto con los objetos de la escena, para posicionarlos a partir del origen del SRU y mirando en la dirección negativa del eje Z. El área sombreada de la cámara se corresponde con el volumen de visualización de la misma (sólo los objetos que estén contenidos en esa pirámide serán finalmente representados).

En la sección 8.2.4 estudiaremos cómo se realiza la proyección en perspectiva de un modo simplificado. Cuando veamos en el capítulo 9, determinaremos la expresión mediante la que los objetos de la escena se proyectan en un volumen simple (el *cubo unitario*) antes de proceder al recorte y su posterior *rasterización*.

Tras la proyección, el *volumen de visualización* se transforma en **Coordenadas Normalizadas** (obteniendo el *cubo unitario*), donde los modelos son proyectados de 3D a 2D. La coordenada Z se guarda habitualmente en un buffer de profundidad llamado *Z-Buffer*.

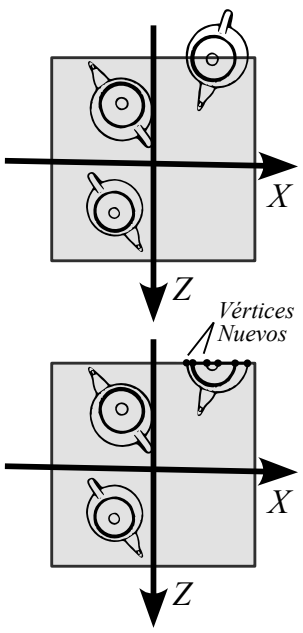


Figura 8.6: Los objetos que intersecan con los límites del *cubo unitario* (arriba) son recortados, añadiendo nuevos vértices. Los objetos que están totalmente dentro del cubo unitario se pasan directamente a la siguiente etapa. Los objetos que están totalmente fuera del *cubo unitario* son descartados.

Únicamente los objetos que están dentro del *volumen de visualización* deben ser generados en la imagen final. Los objetos que están *totalmente* dentro del volumen de visualización serán copiados íntegramente a la siguiente etapa del *pipeline*. Sin embargo, aquellos que estén parcialmente incluidas necesitan ser recortadas, generando nuevos vértices en el límite del recorte. Esta operación de **Transformación de Recorte** se realiza automáticamente por el hardware de la tarjeta gráfica. En la Figura 8.6 se muestra un ejemplo simplificado de recorte.

Finalmente la **Transformación de Pantalla** toma como entrada las coordenadas de la etapa anterior y produce las denominadas **Coordenadas de Pantalla**, que ajustan las coordenadas x e y del cubo unitario a las dimensiones de ventana finales.

8.2.3. Etapa Rasterización

A partir de los vértices proyectados (en *Coordenadas de Pantalla*) y la información asociada a su sombreado obtenidas de la etapa anterior, la etapa de *rasterización* se encarga de calcular los colores finales que se asignarán a los píxeles de los objetos. Esta etapa de rasterización se divide normalmente en las siguientes etapas funciones para lograr mayor paralelismo.

En la primera etapa del pipeline llamada **Configuración de Triángulos** (*Triangle Setup*), se calculan las coordenadas 2D que definen el contorno de cada triángulo (el primer y último punto de cada vértice). Esta información es utilizada en la siguiente etapa (y en la interpolación), y normalmente se implementa directamente en hardware dedicado.

A continuación, en la etapa del **Recorrido de Triángulo** (*Triangle Traversal*) se generan *fragmentos* para la parte de cada píxel que pertenece al triángulo. El recorrido del triángulo se basa por tanto en encontrar los píxeles que forman parte del triángulo, y se denomina *Triangle Traversal* (o *Scan Conversion*). El *fragmento* se calcula interpolando la información de los tres vértices denifidos en la etapa de *Configuración de Triángulos* y contiene información calculada sobre la profundidad desde la cámara y el sombreado (obtenida en la etapa de geometría a nivel de todo el triángulo).

La información interpolada de la etapa anterior se utiliza en el **Pixel Shader** (*Sombreado de Píxel*) para aplicar el sombreado a nivel de píxel. Esta etapa habitualmente se ejecuta en núcleos de la GPU programables, y permite implementaciones propias por parte del usuario. En esta etapa se aplican las texturas empleando diversos métodos de proyección (ver Figura 8.7).

Finalmente en la etapa de **Fusión** (*Merging*) se almacena la información del color de cada píxel en un array de colores denominado *Color Buffer*. Para ello, se combina el resultado de los *fragmentos* que son visibles de la etapa de *Sombreado de Píxel*. La visibilidad se suele resolver en la mayoría de los casos mediante un buffer de profundidad *Z-Buffer*, empleando la información que almacenan los *fragmentos*.

El **Z-Buffer** es un buffer ampliamente empleado en gráficos por computador. Tiene el mismo tamaño en píxeles que el buffer de color, pero almacena la *menor* distancia para cada píxel a todos los fragmentos de la escena. Habitualmente se representa como una imagen en escala de grises, y asocia valores más cercanos a blanco a distancias menores.

8.2.4. Proyección en Perspectiva

En gráficos por computador es posible elegir entre diferentes modelos de proyección de los objetos sobre el plano de visualización. Un modo muy utilizado en aplicaciones de CAD es la proyección de los objetos empleando líneas paralelas sobre el plano de proyección, mediante la denominada **proyección paralela**. En este modo de proyección se conservan las proporciones relativas entre objetos, independientemente de su distancia.

Mediante la **proyección en perspectiva** se proyectan los puntos hasta el plano de visualización empleando trayectorias convergentes en un punto. Esto hace que los objetos situados más distantes del plano de visualización aparezcan más pequeños en la imagen. Las escenas generadas utilizando este modelo de proyección son más realistas, ya que ésta es la manera en que el ojo humano y las cámaras físicas forman imágenes.

En la proyección en perspectiva, las líneas paralelas convergen en un punto, de forma que los objetos más cercanos se muestran de un tamaño mayor que los lejanos. Desde el 500aC, los griegos estudiaron el fenómeno que ocurría cuando la luz pasaba a través de pequeñas aberturas. La primera descripción de una cámara estenopeica se atribuye al atrónomo y matemático holandés *Gemma Frisius* que en 1545 publicó la primera descripción de una *cámara oscura* en la observación de un eclipse solar (ver Figura 8.8). En las cámaras esteneopeicas la luz pasa a través de un pequeño agujero para formar la imagen en la película fotosensible, que aparece invertida. Para que la imagen sea nítida, la abertura debe ser muy pequeña.

Siguiendo la misma idea y desplazando el plano de proyección delante del origen, tenemos el modelo general proyección en perspectiva.

Consideraremos en el resto de la sección que ya se ha realizado la *transformación de visualización* alineando la cámara y los objetos de la escena mirando en dirección al eje negativo Z, que el eje Y está apuntando hacia arriba y el eje X positivo a la derecha (como se muestra en la Figura 8.5).

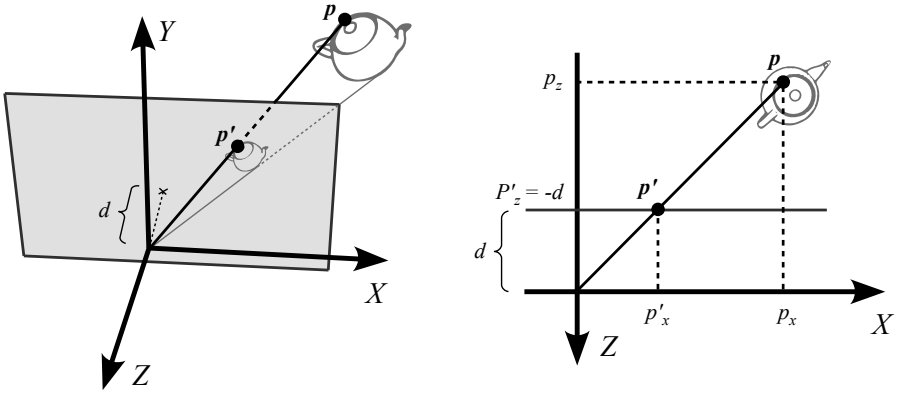


Figura 8.9: Modelo de proyección en perspectiva simple. El plano de proyección infinito está definido en $z = -d$, de forma que el punto p se proyecta sobre p'.

En la Figura 8.9 se muestra un ejemplo de proyección simple, en la que los vértices de los objetos del mundo se proyectan sobre un plano infinito situado en $z = -d$ (con $d > 0$). Suponiendo que la *transformación de visualización* se ha realizado, proyectamos un punto p sobre el plano de proyección, obteniendo un punto $p' = (p'_x, p'_y, -d)$.

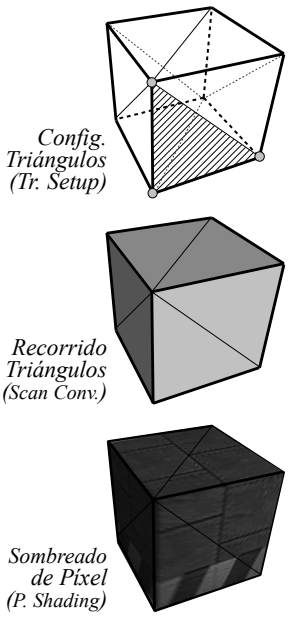


Figura 8.7: Representación del resultado de las principales etapas del Pipeline de Rasterización.

Perspectiva

La mayoría de los juegos 3D se basan en el uso de cámaras de proyección en perspectiva. Estudiaremos con más detalle este tipo de modelos de proyección en el capítulo 9.

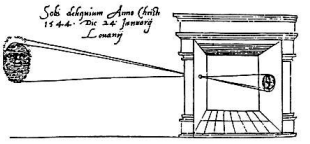


Figura 8.8: Descripción de la primera cámara estenopeica (*pinhole camera* o *camera obscura*) por Gemma Frisius.

Empleando triángulos semejantes (ver Figura 8.9 derecha), obtenemos las siguientes coordenadas:

$$\frac{p'_x}{p_x} = \frac{-d}{p_z} \quad \Leftrightarrow \quad p'_x = \frac{-d\,p_x}{p_z} \tag{8.1}$$

De igual forma obtenemos la coordenada $p'_y = -d\,p_y/p_z$, y $p'_z = -d$. Como veremos en el Capítulo 9, estas ecuaciones se pueden expresar fácilmente de forma matricial (que es la forma habitual de trabajar internamente en el pipeline). Estudiaremos más detalles sobre el modelo de proyección en perspectiva en el Capítulo 9, así como la transformación que se realiza de la pirámide de visualización (*Frustum*) al cubo unitario.

8.3. Implementación del Pipeline en GPU

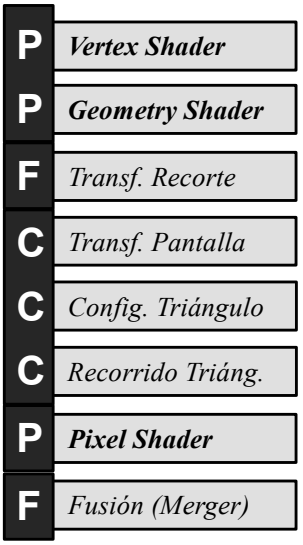


Figura 8.10: Implementación típica del pipeline en GPU con soporte Híbrido (OpenGL 2). La letra asociada a cada etapa indica el nivel de modificación permitida al usuario; **P** indica totalmente programable, **F** indica fijo (no programable), y **C** indica configurable pero no programable.

El hardware de aceleración gráfica ha sufrido una importante transformación en la última década. Como se muestra en la Figura 8.11, en los últimos años el potencial de las GPUs ha superado con creces al de la CPU.

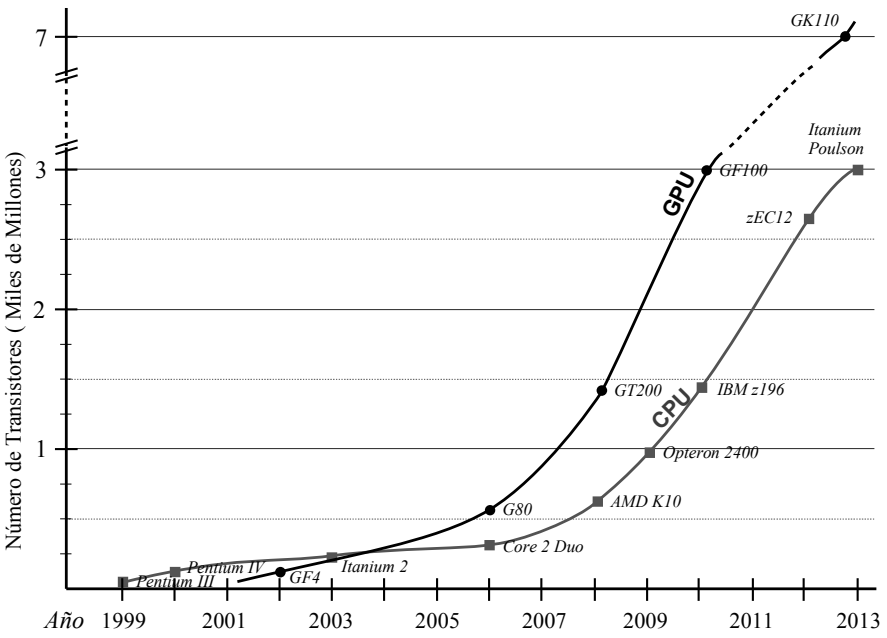


Figura 8.11: Evolución del número de transistores en GPU y CPU (1999-2013)

Resulta de especial interés conocer aquellas partes del Pipeline de la GPU que puede ser programable por el usuario mediante el desarrollo de *shaders*. Este código se ejecuta directamente en la GPU, y permite realizar operaciones a diferentes niveles con alta eficiencia.

En GPU, las etapas del *Pipeline gráfico* (estudiadas en la sección 8.2) se implementan en un conjunto de etapas diferente, que además pueden o no ser programables por parte del usuario (ver Figura 8.10). Por cuestiones de eficiencia, algunas partes del Pipeline en GPU no son programables, aunque se prevee que la tendencia en los próximos años sea permitir su modificación.

La etapa asociada al *Vertex Shader* es totalmente programable, y encapsula las cuatro primeras etapas del pipeline gráfico que estudiamos en la sección 8.2: *Transformación de Modelado*, *Transformación de Visualización*, *Sombreado de Vértices (Vertex Shader)* y *Transformación de Proyección*.

La etapa referente al *Geometry Shader* es otra etapa programable que permite realizar operaciones sobre las primitivas geométricas.

El término GPU

Desde el 1999, se utiliza el término GPU (*Grpahics Processing Unit*) acuñado por NVIDIA para diferenciar la primera tarjeta gráfica que permitía al programador implementar sus propios algoritmos (*GeForce 256*).

La evolución natural de las dos plataformas principales de gráficos 3D en tiempo real (tanto OpenGL como Direct3D) ha sido el soporte único de etapas totalmente programables. Desde 2009, OpenGL en su versión 3.2 incluye un modo *Compatibility Profile* manteniendo el soporte de llamadas a las etapas no programables. De forma similar, en 2009 Direct 3D versión 10 eliminó las llamadas a las etapas fijas del pipeline.

Las etapas de *Transformación de Recorte*, *Transformación de Pantalla*, *Configuración de Triángulo*, *Recorrido de Triángulo* y *Fusión* tienen un comportamiento funcional similar al estudiado en la sección 8.2, por lo que no serán descritas de nuevo.

Finalmente, el *Pixel Shader* es la última etapa totalmente programable del pipeline en GPU y permite al programador desarrollar operaciones específicas a nivel de píxel.

El desarrollo de shaders se ha incorporado desde el 2009 en las especificaciones de las pricipales APIs gráficas (OpenGL y Direct3D). Estas llamadas son compiladas a un lenguaje ensamblador intermedio independiente de la tarjeta gráfica. Son los drivers de cada tarjeta gráfica los que transforman este lenguaje intermedio en instrucciones específicas para cada tarjeta.

Veremos a continuación brevemente algunas características de estas etapas programables de la GPU.

Ensamblador?

En realidad este código ensamblador intermedio puede verse como una especie de código de máquina virtual que garantiza la compatibilidad entre diferentes dispositivos hardware.

8.3.1. Vertex Shader

Los *vertex shaders* permiten aplicar transformaciones y deformaciones a nivel de vértice. Este *shader* se aplica en la primera etapa del pipeline de la GPU. En esta etapa, los flujos de datos que la CPU envía a la tarjeta son procesados y se aplican las matrices de transformación especificadas por el usuario.

En esta etapa se aplican las instancias sobre los datos enviados a la GPU (evitando enviar varias veces las mismas primitivas geométricas). A este nivel, el *vertex shader* únicamente trabaja con la información relativa a los vértices (posición, vector normal, color y coordenadas de textura). El *vertex shader* no conoce nada sobre la conexión de estos vértices entre sí para formar triángulos.

Algunas operaciones clásicas que se implementan empleando *vertex shader* son efectos de lente (como por ejemplo, de ojo de pez o distorsiones como las causadas en escenas submarinas), deformaciones de objetos, animaciones de textura, etc.

8.3.2. Geometry Shader

Los *geometry shaders* facilitan la creación y destrucción de primitivas geométricas en la GPU en tiempo de ejecución (vértices, líneas y triángulos).

La entrada de este módulo lo forman la especificación de los objetos con sus vértices asociados. Para cada primitiva de entrada, el *geometry shader* devolverá cero o más primitivas de salida. Las primitivas de entrada y salida no tienen por qué ser del mismo tipo. Por ejemplo, es posible indicar un triángulo como entrada (tres vértices 3d) y devolver el centroide (un punto 3D) como salida. Las primitivas del flujo de salida del *geometry shader* se obtienen en el mismo orden que se especificaron las primitivas de entrada.

Este tipo de shaders se emplean para la simulación de pelo, para encontrar los bordes de los objetos, o para implementar algunas técnicas de visualización avanzadas como metabolas o simulación de telas.

8.3.3. Pixel Shader

A nivel de *pixel shader* se pueden aplicar operaciones a nivel de píxel, permitiendo definir complejas ecuaciones de sombreado que serán evaluadas para cada píxel de la imagen.

El *Pixel Shader* tiene influencia únicamente sobre el fragmento que está manejando. Esto implica que no puede aplicar ninguna transformación sobre fragmentos vecinos.

El uso principal que se da a este tipo de *shaders* es el establecimiento mediante código del color y la profundidad asociada al fragmento. Actualmente se emplea para aplicar multitud de efectos de representación no realista, reflexiones, etc.

Figura 8.12: Resultado de aplicar un *Vertex Shader* para deformar un modelo.

Notación...

En OpenGL al *Pixel Shader* se le denomina *Fragment Shader*. En realidad es un mejor nombre, porque se trabaja a nivel de *fragmento*.

8.4. Arquitectura del motor gráfico

El objetivo de esta sección es proporcionar una primera visión general sobre los conceptos generales subyacentes en cualquier motor gráfico 3D interactivo. Estos conceptos serán estudiados en profundidad a lo largo de este documento, mostrando su uso práctico mediante ejemplos desarrollados en C++ empleando el motor gráfico OGRE.

Como se ha visto en la introducción del capítulo, los videojuegos requieren hacer un uso eficiente de los recursos gráficos. Hace dos décadas, los videojuegos se diseñaban específicamente para una plataforma hardware específica, y las optimizaciones podían realizarse a muy bajo nivel. Actualmente, el desarrollo de un videojuego tiende a realizarse para varias plataformas, por lo que el uso de un motor gráfico que nos abstraiga de las particularidades de cada plataforma no es una opción, sino una necesidad.

Compatiblidad

En algunos casos, como en el desarrollo de videojuegos para PC, el programador no puede hacer casi ninguna suposición sobre el hardware subyacente en la máquina donde se ejecutará finalmente el programa. En el caso de desarrollo para consolas, los entornos de ejecución concretos están mucho más controlados.

En estos desarrollos multiplataforma es necesario abordar aproximaciones de diseño que permitan emplear diferentes *perfiles* de ejecución. Por ejemplo, en máquinas con grandes prestaciones se emplearán efectos y técnicas de despliegue más realistas, mientras que en máquinas con recursos limitados se utilizarán algoritmos con menores requisitos computacionales y versiones de los recursos gráficos adaptadas (con diferente nivel de detalle asociado).

Las limitaciones asociadas a los recursos computacionales son una constante en el área del desarrollo de videojuegos. Cada plataforma conlleva sus propias limitaciones y restricciones, que pueden asociarse en las categorías de:

Figura 8.13: Capturas de algunos videojuegos desarrollados con motores libres. La imagen de la izquierda se corresponde con *Planeshift*, realizado con Crystal Space. La captura de la derecha es del videojuego *H-Craft Championship*, desarrollado con Irrlicht.

- **Tiempo de Procesamiento**. El desarrollo de videojuegos en prácticamente cualquier plataforma actual require el manejo de múltiples núcleos de procesamiento (tanto de CPU como GPU). El manejo explícito de la concurrencia (habitualmente a nivel de hilos) es necesario para mantener una alta tasa de *Frames por Segundo*.
- **Almacenamiento**. En el caso de ciertos dispositivos como consolas, la variedad de unidades de almacenamiento de los recursos del juego (con velocidades de acceso y transferencia heterogéneas), dificultan el desarrollo del videojuego. En ciertas plataformas, no se dispone de aproximaciones de memoria virtual, por lo que el programador debe utilizar explícitamente superposiciones (*overlays*) para cargar las zonas del juego que van a emplearse en cada momento.

Dada la gran cantidad de restricciones que deben manejarse, así como el manejo de la heterogeneidad en términos software y hardware, es necesario el uso de un motor de despliegue gráfico para desarrollar videojuegos. A continuación enunciaremos algunos de los más empleados.

8.5. Casos de Estudio

En esta sección se estudiarán algunos de los motores gráficos 3D libres, comentando brevemente algunas de sus características más destacables.

- **Crystal Space**. Crystal Space (`http://www.crystalspace3d.org/`) es un framework completo para el desarrollo de videojuegos escrito en C++, desarrollado inicialmente en 1997. Se distribuye bajo licencia libre LGPL, y es multiplataforma (GNU/Linux, Windows y Mac).
- **Panda 3D**. Panda 3D (`http://www.panda3d.org/`) es un motor para el desarrollo de videojuegos multiplataforma. Inicialmente fue desarrollado por Disney para la construcción del software asociado a las atracciones en parques temáticos, y posteriormente liberado en 2002 bajo licencia BSD. Este motor mul-

tiplataforma (para GNU/Linux, Windows y Mac) incluye interfaces para C++ y Python. Su corta curva de aprendizaje hace que haya sido utilizado en varios cursos universitarios, pero no ofrece características de representación avanzadas y el interfaz de alto nivel de Python conlleva una pérdida de rendimiento.

- **Irrlicht**. Este motor gráfico de renderizado 3D, con primera versión publicada en el 2003 (`http://irrlicht.sourceforge.net/`), ofrece interfaces para C++ y .NET. Existen gran cantidad de *wrappers* a diferentes lenguajes como Java, Perl, Python o Lua. Irrlicht tiene una licencia Open Source basada en la licencia de ZLib. Irrlicht es igualmente multiplataforma (GNU/Linux, Windows y Mac).
- **OGRE**. OGRE (`http://www.ogre3d.org/`) es un motor para gráficos 3D libre multiplataforma. Sus características serán estudiadas en detalle en la sección 8.6.

De entre los motores estudiados, OGRE 3D ofrece una calidad de diseño superior, con características técnicas avanzadas que han permitido el desarrollo de varios videojuegos comerciales. Además, el hecho de que se centre exclusivamente en el capa gráfica permite utilizar gran variedad de bibliotecas externas (habitualmente accedidas mediante *plugins*) para proporcionar funcionalidad adicional. En la siguiente sección daremos una primera introducción y toma de contacto al motor libre OGRE.

Figura 8.14: El logotipo de OGRE 3D es un... OGRO!

¡Sólo Rendering!

El desarrollo de OGRE se centra exclusivamente en la parte de despliegue gráfico. El motor no proporciona mecanismos para capturar la interacción del usuario, ni para reproducción audio o gestión del estado interno del videojuego.

8.6. Introducción a OGRE

OGRE es un motor orientado a objetos libre para aplicaciones gráficas 3D interactivas. El nombre del motor OGRE es un acrónimo de *Object-oriented Graphics Rendering Engine*. Como su propio nombre indica, OGRE no es un motor para el desarrollo de videojuegos; se centra exclusivamente en la definición de un *middleware* para el renderizado de gráficos 3D en tiempo real.

El proyecto de OGRE comenzó en el 2000 con el propósito de crear un motor gráfico bien diseñado. El líder del proyecto Steve Streeting define el desarrollo de OGRE como un proyecto basado en la *calidad* más que en la *cantidad* de características que soporta, porque la cantidad viene con el tiempo, y la calidad nunca puede añadirse a posteriori. La popularidad de OGRE se basa en los principios de *meritocracia* de los proyectos de software libre. Así, el sitio web de OGRE [2] recibe más de 500.000 visitas diarias, con más de 40.000 descargas mensuales.

La última versión la biblioteca en desarrollo de la biblioteca (1.9) ha sido publicada en Abril de 2013 [3].

El núcleo principal de desarrolladores en OGRE se mantiene deliberadamente pequeño y está formado por profesionales con dilatada experiencia en proyectos de ingeniería reales.

OGRE tiene una licencia LGPL *Lesser GNU Public License*. Esta licencia se utiliza con frecuencia en bibliotecas que ofrecen funcionalidad que es similar a la de otras bibliotecas privativas. Por cuestión de estrategia, se publican bajo licencia LGPL (o GPL *Reducida*) para permitir que se enlacen tanto por programas libres como no libres. La única restricción que se aplica es que si el enlazado de las bibliotecas es estático, la aplicación resultado debe ser igualmente LGPL (porque el *enlazado estático* también *enlaza la licencia*).

[2] http://www.ogre3d.org

[3] En el curso utilizaremos la última versión estable (Ogre 1.8)

La versión oficial de OGRE está desarrollada en C++ (el lenguaje estándar en el ámbito del desarrollo de videojuegos). La rama oficial de OGRE únicamente se centra en este lenguaje sobre los sistemas operativos GNU/Linux, Mac OS X y Microsoft Windows. No obstante, existen *wrappers* de la API a otros lenguajes (como Java, Python o C#) que son mantenidos por la comunidad de usuarios (presentando diferentes niveles de estabilidad y completitud), que no forman parte del núcleo oficial de la biblioteca.

En el momento de la escritura de este texto, la última versión en desarrollo 1.9RC1 de Ogre (*inestable*), incluye soporte para nuevas plataformas como Android, Windows Phone 8, así como la escritura del motor con soporte para OpenGL 3, DirectX11, etc...

Algunas características destacables de OGRE son:

Portabilidad

Uno de los objetivos de diseño de OGRE es utilizar otras bibliotecas multiplataforma estables en su desarrollo, como *FreeType* para el despliegue de fuentes TrueType, *OpenIL* para la carga y manipulación de imágenes y *ZLib* para la gestión de archivos comprimidos ZIP.

- **Motor Multiplataforma**. Aunque el desarrollo original de OGRE se realizó bajo plataformas Microsoft Windows, la distribución oficial ofrece versiones binarias para GNU/Linux y Mac OS X. Además, gracias al soporte nativo de OpenGL, es posible compilar la biblioteca en multitud de plataformas (como diversas versiones de Unix, además de algunos *ports* no oficiales para Xbox y dispositivos portátiles). OGRE soporta la utilización de las APIs de despliegue gráfico de bajo nivel OpenGL y Direct3D.

- **Diseño de Alto Nivel**. OGRE encapsula la complejidad de acceder directamente a las APIs de bajo nivel (como OpenGL y Direct3D) proporcionando métodos intuitivos para la manipulación de objetos y sus propiedades relacionadas. De este modo no es necesario gestionar manualmente la geometría o las matrices de transformación. Todos los objetos representables de la escena se abstraen en un interfaz que encapsula las operaciones necesarias para su despliegue (técnicas empleadas y composición).

  OGRE hace un uso de varios **patrones de diseño** para mejorar la usabilidad y la flexibilidad de la bibloteca. Por ejemplo, para informar a la aplicación sobre eventos y cambios de estado utiliza el patrón *Observador*. El patrón *Singleton* se emplea en gran número de *Gestores* para forzar que únicamente exista una instancia de una clase. El patrón *Visitor* se emplea para permitir operaciones sobre un objeto sin necesidad de modificarlo (como en el caso de los nodos del grafo de escena), el patrón *Facade* para unificar el acceso a operaciones, *Factoría* para la creación de instancias concretas de interfaces abstractos, etc. Estos patrones se estudian en detalle en el Módulo 1 del curso.

- **Grafos de Escena**. Prácticamente cualquier biblioteca de despliegue de gráficos 3D utiliza un *Grafo de Escena* para organizar los elementos que serán representados en la misma. Un objetivo fundamental en el diseño de esta estructura de datos es permitir búsquedas eficientes. Una de las características más potentes de OGRE es el desacople del grafo de escena del contenido de la escena, definiendo una arquitectura de plugins. En pocas palabras, a diferencia de otros motores gráficos como Irrlicht3D, Blitz3D o TrueVision3D (o motores de videojuegos como Torque, CryEngine o Unreal), OGRE no se basa en la *Herencia* como principio de diseño del Grafo de Escena, sino en la *Composición*. Esto permite expandir el diseño cómodamente para soportar otros tipos de datos (como audio o elementos de simulación física).

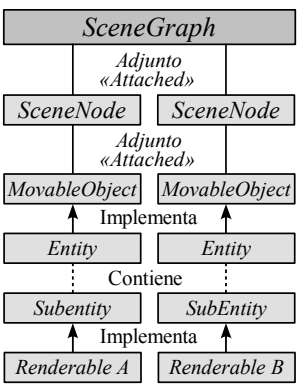


Figura 8.15: Esquema general de la gestión del grafo de escena en OGRE.

La Figura 8.15 muestra el esquema general de gestión del grafo de escena en OGRE. Los *Renderables* manejan la geometría de la escena. Todas las propiedades para el despliegue de estos *Renderables* (como por ejemplo los materiales) se gestionan en objetos de tipo Entidad (*Entity*) que pueden estar formados por varios objetos SubEntidad (*SubEntity*). Como se ha comentado anteriormente, la escena se *Compone* de nodos de escena. Estos *SceneNodes* se adjuntan a la escena. Las propiedades de esos nodos de escena (geometría, materiales, etc...) se ofrecen al *SceneGraph* mediante un *MovableObject*. De forma similar, los *MovableObjects* no son subclases de *SceneNode*, sino que se adjuntan. Esto permite realizar modificaciones en la implementación del grafo de escena sin necesidad de tocar ninguna línea de código en la implementación de los objetos que contiene. Veremos más detalles sobre el trabajo con el grafo de escena a lo largo de este módulo.

- **Aceleración Hardware**. OGRE necesita una tarjeta aceleradora gráfica para poder ejecutarse (con soporte de *direct rendering mode*). OGRE permite definir el comportamiento de la parte programable de la GPU mediante la definición de *Shaders*, estando al mismo nivel de otros motores como Unreal o CryEngine.

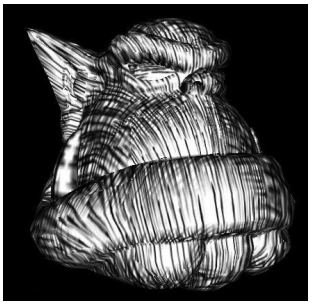


Figura 8.16: Ejemplo de Shader desarrollado por Assaf Raman para simular trazos a Lápiz empleando el sistema de materiales de OGRE.

- **Materiales**. Otro aspecto realmente potente en OGRE es la gestión de materiales. Es posible crear materiales sin modificar ni una línea de código a compilar. El sistema de *scripts* para la definición de materiales de OGRE es uno de los más potentes existentes en motores de rendering interactivo. Los materiales de OGRE se definen mediante una o más **Técnicas**, que se componen a su vez de una o más **Pasadas** de rendering (el ejemplo de la Figura 8.16 utiliza una única pasada para simular el sombreado a lápiz mediante *hatching*). OGRE busca automáticamente la mejor técnica disponible en un material que esté soportada por el hardware de la máquina de forma transparente al programador. Además, es posible definir diferentes **Esquemas** asociados a modos de calidad en el despliegue.

- **Animación**. OGRE soporta tres tipos de animación ampliamente utilizados en la construcción de videojuegos: basada en esqueletos (*skeletal)*, basada en vértices (*morph* y *pose*). En la animación mediante esqueletos, OGRE permite el uso de esqueletos con animación basada en cinemática directa. Existen multitud de exportadores para los principales paquetes de edición 3D. En este módulo utilizaremos el exportador de Blender.

  El sistema de animación de OGRE se basa en el uso de **controladores**, que modifican el valor de una propiedad en función de otro valor. En el caso de animaciones se utiliza el *tiempo* como valor para modificar otras propiedades (como por ejemplo la posición del objeto). El motor de OGRE soporta dos modelos básicos de interpolación: lineal y basada en splines cúbicas.

  La animación y la geometría asociada a los modelos se almacena en un único formato binario optimizado. El proceso más empleado se basa en la exportación desde la aplicación de modelado y animación 3D a un formato XML (*Ogre XML*) para convertirlo posteriormente al formato binario optimizado mediante la herramienta de línea de órdenes `OgreXMLConverter`.

Optimización offline

El proceso de optimización de al formato binario de OGRE calcula el orden adecuado de los vértices de la malla, calcula las normales de las caras poligonales, así como versiones de diferentes niveles de detalle de la malla. Este proceso evita el cálculo de estos parámetros en tiempo de ejecución.

- **Composición y Postprocesado**. El framework de composición facilita al programador incluir efectos de postprocesado en tiempo de ejecución (siendo una extensión del *pixel shader* del pipeline estudiado en la sección 8.3.3). La aproximación basada en *pasadas* y diferentes *técnicas* es similar a la explicada para el gestor de materiales.

- **Plugins**. El diseño de OGRE facilita el diseño de *Plugins* como componentes que cooperan y se comunican mediante un interfaz conocido. La gestión de archivos, sistemas de rendering y el sistema de partículas están implementados basados en el diseño de Plugins.

- **Gestión de Recursos**. Para OGRE los recursos son los elementos necesarios para representar un objetivo. Estos elementos son la geometría, materiales, esqueletos, fuentes, scripts, texturas, etc. Cada uno de estos elementos tienen asociado un gestor de recurso específico. Este gestor se encarga de controlar la cantidad de memoria empleada por el recurso, y facilitar la carga, descarga, creación e inicialización del recurso. OGRE organiza los recursos en niveles de gestión superiores denominados *grupos*. Veremos en la sección 8.6.1 los recursos gestionados por OGRE.
- **Características específicas avanzadas**. El motor soporta gran cantidad de características de visualización avanzadas, que estudiaremos a lo largo del módulo, tales como sombras dinámicas (basadas en diversas técnicas de cálculo), sistemas de partículas, animación basada en esqueletos y de vértices, y un largo etcétera. OGRE soporta además el uso de otras bibliotecas auxiliares mediante plugins y conectores. Entre los más utilizados cabe destacar las bibliotecas de simulación física ODE, el soporte del metaformato Collada, o la reproducción de streaming de vídeo con Theora. Algunos de estos módulos los utilizaremos en el módulo 3 del presente curso.

8.6.1. Arquitectura General

El diagrama de la Figura 8.17 resume algunos de los objetos principales del motor OGRE. No es un diagrama exhaustivo, pero facilita la comprensión de los principales módulos que utilizaremos a lo largo del curso.

Objeto Root

El objeto Root es el eje pricipal sobre el que se define una aplicación que utiliza OGRE. La creación de una instancia de esta clase hará que se inicie OGRE, y su destrucción hará que se libere la memoria asociada a todos los objetos que dependen de él.

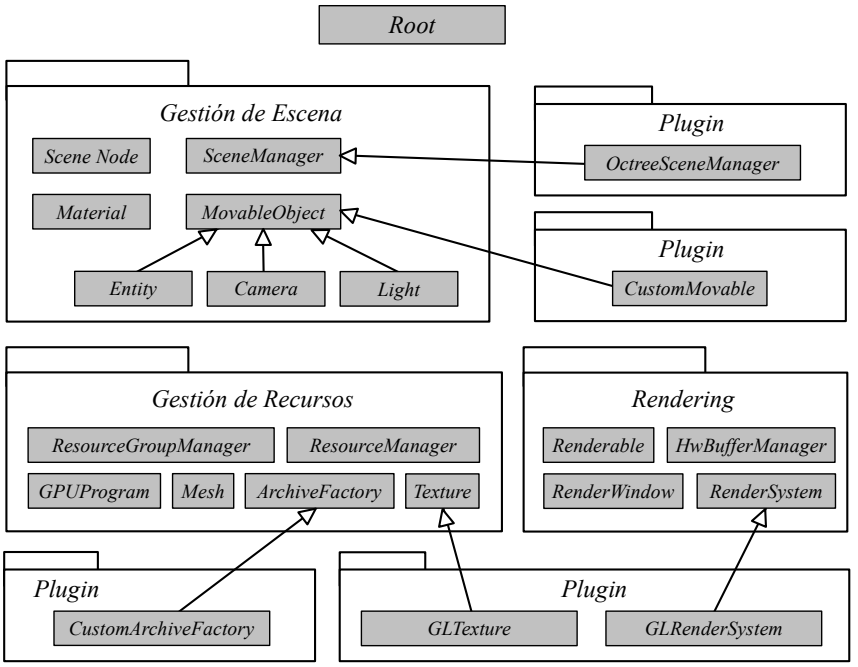


Figura 8.17: Diagrama general de algunos de los objetos principales de OGRE

Uno de los objetos principales del sistema es el denominado *Root*. Root proporciona mecanismos para la creación de los objetos de alto nivel que nos permitirán gestionar las escenas, ventanas, carga de plugins, etc. Gran parte de la funcionalidad de Ogre se proporciona a través del objeto *Root*. Como se muestra en el diagrama 8.17, existen otros tres grupos de clases principales en OGRE:

- **Gestión de Escena**: Los objetos de este grupo de clases se encargan de definir el contenido de la escena virtual, su estructura, así como otras propiedades de alto nivel de abstracción (posición de la cámara, posición de los objetos, materiales, etc...). Como se estudió en la Figura 8.15, el grafo de escena es un elemento principal en la arquitectura de cualquier motor 3D. En el caso de OGRE, el Gestor de Escena (clase *SceneManager*) es el que se encarga de implementar el grafo de escena. Los nodos de escena *SceneNode* son elementos relacionados jerárquicamente, que pueden adjuntarse o desligarse de una escena en tiempo de ejecución. El contenido de estos *SceneNode* se adjunta en la forma de instancias de Entidades (*Entity*), que son implementaciones de la clase *MovableObject*.
- **Gestión de Recursos**: Este grupo de objetos se encarga de gestionar los recursos necesarios para la representación de la escena (geometría, texturas, tipografías, etc...). Esta gestión permite su carga, descarga y reutilización (mediante cachés de recursos). En la siguiente subsección veremos en detalle los principales gestores de recursos definidos en OGRE.
- **Rendering**: Este grupo de objetos sirve de *intermediario* entre los objetos de *Gestión de Escena* y el pipeline gráfico de bajo nivel (con llamadas específicas a APIs gráficas, trabajo con buffers, estados internos de despliegue, etc.). La clase *RenderSystem* es un interfaz general entre OGRE y las APIs de bajo nivel (OpenGL o Direct3D). La forma más habitual de crear la *RenderWindow* es a través del objeto *Root* o mediante el *RenderSystem* (ver Figura 8.18). La creación manual permite el establecimiento de mayor cantidad de parámetros, aunque para la mayoría de las aplicaciones la creación automática es más que suficiente.

Entidades Procedurales

La mayor parte de las entidades que incluyas en el *SceneNode* serán cargadas de disco (como por ejemplo, una malla binaria en formato `.mesh`). Sin embargo, OGRE permite la definción en código de otras entidades, como una textura procedural, o un plano.

Por ser la gestión de recursos uno de los ejes principales en la creación de una aplicación gráfica interactiva, estudiaremos a continuación los grupos de gestión de recursos y las principales clases relacionadas en OGRE.

Gestión de Recursos

Como hemos comentado anteriormente, cualquier elemento necesario para represesntar una escena se denomina *recurso*. Todos los recursos son gestionados por un único objeto llamado *ResourceGroupManager*, que se encarga de buscar los recursos definidos en la aplicación e inicializarlos. Cada tipo de recurso tiene asociado un gestor de recurso particular. Veamos a continuación los tipos de recursos soportados en OGRE:

- **Mallas**. La geometría de los elementos de la escena se especifica en un formato de malla binario (`.mesh`). Este formato almacena la geometría y la animación asociada a los objetos.
- **Materiales**. Como se ha descrito anteriormente, el material se especifica habitualmente mediante scripts (en ficheros de extensión `.material`). Estos scripts pueden estar referenciados en el propio archivo `.mesh` o pueden ser enlazados a la malla mediante código compilado.

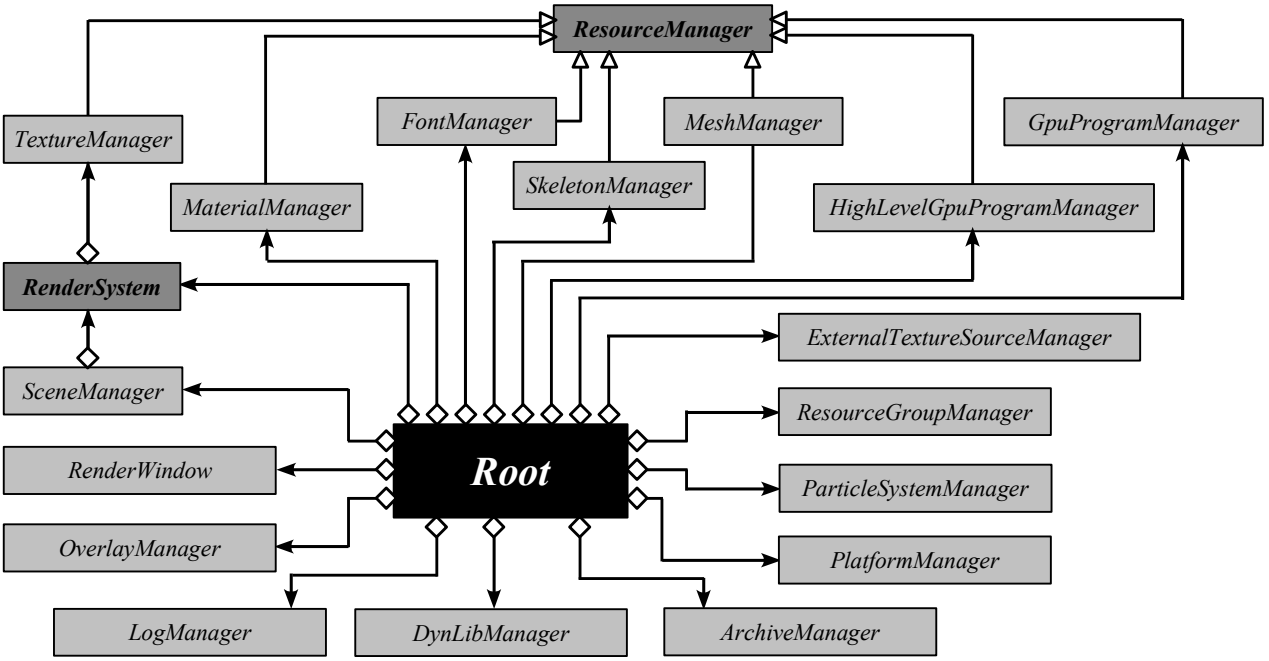


Figura 8.18: Diagrama de clases simplificado de los *Gestores* de alto nivel que dan acceso a los diferentes subsistemas definidos en OGRE

- **Texturas**. OGRE soporta todos los formatos de texturas 2D admitidos por la biblioteca OpenIL. El formato se reconoce por la extensión asociada al mismo.
- **Esqueletos**. Habitualmente el esqueleto está referenciado en el fichero `.mesh`. El fichero de definición de esqueleto contiene la jerarquía de huesos asociada al mismo y tiene una extensión `.skeleton`.
- **Fuentes**. Las fuentes empleadas en la etapa de despliegue de *Overlays* se definen en un archivo `.fontdef`.
- **Composición**. El framework de composición de OGRE carga sus scripts con extensión `.compositor`.
- **GPU**. El código de shaders definidos para la GPU (de HLSL, GLSL y Cg) se describe en archivos con extensión `.program`. De igual modo se pueden definir código en ensamblador mediante archivos `.asm`. Los programas de GPU son cargados antes que se procese cualquier archivo de material `.material`, de forma que estos shaders puedan ser referenciados en los archivos de definición de material.

Un gestor (*Manager*) en OGRE es una clase que gestiona el acceso a otros tipos de objetos (ver Figura 8.18). Los *Managers* de OGRE se implementan mediante el patrón *Singleton* que garantiza que únicamente hay una istancia de esa clase en toda la aplicación. Este patrón permite el acceso a la instancia *única* de la clase *Manager* desde cualquier lugar del código de la aplicación.

Como puede verse en la Figura 8.18, el *ResourceManager* es una clase abstracta de la que heredan un gran número de *Managers*, tales como el encargado de las Fuentes, las Texturas o las Mallas. A continuación se ofrece una descripción general de los *Managers* (por orden alfabético) definidos en OGRE. A lo largo del documento se describirán (y se utilizarán ampliamente) muchos de estos gestores.

- **Archive Manager**. Se encarga de abstraer del uso de ficheros con diferentes extensiones, directorios y archivos empaquetados `.zip`.
- **CompositorManager**. Esta clase proporciona acceso al framework de composición y postprocesado de OGRE.
- **ControllerManager**. Es el gestor de los *controladores* que, como hemos indicado anteriormente, se encargan de producir cambios en el estado de otras clases dependiendo del valor de ciertas entradas.
- **DynLibManager**. Esta clase es una de las principales en el diseño del sistema de Plugins de OGRE. Se encarga de gestionar las bibliotecas de enlace dinámico (DLLs en Windows y objetos compartidos en GNU/Linux).
- **ExternalTextureSourceManager**. Gestiona las fuentes de textura externas (como por ejemplo, en el uso de *video streaming*).
- **FontManager**. Gestiona las fuentes disponibles para su representación en superposiciones 2D (ver *OverlayManager*).
- **GpuProgramManager**. Carga los programas de alto nivel de la GPU, definidos en ensamblador. Se encarga igualmente de cargar los programas compilados por el *HighLevelGpuProgramManager*.
- **HighLevelGpuProgramManager**. Gestiona la carga y compilación de los programas de alto nivel en la GPU (*shaders*) utilizados en la aplicación en HLSL, GLSL o Cg.
- **LogManager**. Se encarga de enviar mensajes de *Log* a la salida definida por OGRE. Puede ser utilizado igualmente por cualquier código de usuario que quiera enviar eventos de *Log*.
- **MaterialManager**. Esta clase mantiene las instancias de *Material* cargadas en la aplicación, permitiendo reutilizar los objetos de este tipo en diferentes objetos.
- **MeshManager**. De forma análoga al *MaterialManager*, esta clase mantiene las instancias de *Mesh* permitiendo su reutilización entre diferentes objetos.
- **OverlayManager**. Esta clase gestiona la carga y creación de instancias de superposiciones 2D, que permiten dibujar el interfaz de usuario (botones, iconos, números, radar...). En general, los elementos definidos como HUDs (*Head Up Display*).
- **ParticleSystemManager**. Gestiona los sistemas de partículas, permitiendo añadir gran cantidad de efectos especiales en aplicaciones 3D. Esta clase gestiona los emisores, los límites de simulación, etc.
- **PlatformManager**. Esta clase abstrae de las particularidades del sistema operativo y del hardware subyacente de ejecución, proporcionando rutinas independientes del sistema de ventanas, temporizadores, etc.

- **ResourceGroupManager**. Esta clase gestiona la lista de grupos de recursos y se encarga de notificar a los *Managers* de la necesidad de cargar o liberar recursos en cada grupo.
- **SceneManager**. Como se ha explicado anteriormente, esta clase se encarga de la gestión, organización y *rendering* de la escena. Esta clase permite definir subclases que organicen la escena de una forma más eficiente, dependiendo del tipo de aplicación. Por defecto, el *SceneManager* utiliza una jerarquía de cajas límite (*bounding boxes*) para optimizar el despliegue de los objetos.
- **SkeletonManager**. Al igual que el *MaterialManager* y el *MeshManager*, esta clase mantiene las instancias de *Skeleton* cargadas en la aplicación, permitiendo su reutilización entre diferentes objetos.
- **TextureManager**. Gestiona la carga y uso de las texturas de la aplicación.

SceneManager

Los *SceneManagers* de OGRE son implementados como Plugins, de forma que el usuario puede cargar varios gestores de escena en su aplicación. Si el videojuego requiere escenas de interiores con mucha geometría, así como escenas de exteriores, puede ser interesante cargar dos gestores de escena diferentes optimizados para cada parte del juego.

8.6.2. Instalación

En esta sección se detallará el proceso de instalación de la biblioteca OGRE 1.8 en sistemas *GNU/Linux* y *Microsoft Windows*. Nos centraremos en la instalación en distribuciones Debian, que servirán como base para el desarrollo del presente curso. No obstante, se proporcionarán igualmente las herramientas necesarias y makefiles adaptados para Microsoft Windows empleando *MinGW*.

GNU/Linux (Debian)

Para comenzar, instalaremos las herramientas de compilación básicas necesarias para compilar: *gcc*, *g++* y *make* se encuentran disponibles en el metapaquete *build-essential*:

```
apt-get install build-essential
```

A continuación instalaremos los paquetes específicos de OGRE:

```
apt-get install libogre-1.8.0 libogre-1.8-dev ogre-1.8-doc ogre-1.8-
    tools
```

El paquete `libogre-1.8.0` contiene las bibliotecas necesarias para la ejecución de las aplicaciones desarrolladas con OGRE. El paquete `libogre-dev` contiene los ficheros de cabecera instalados en `/usr/include/OGRE` necesarios para compilar nuestros propios ejemplos. El paquete de documentación `ogre-doc` instala en `/usr/share/doc/ogre-doc` la documentación (manual de usuario y API). Finalmente el paquete `ogre-tools` contiene las herramientas para convertir al formato de malla binario optimizado de OGRE.

Como se comentó en la introducción, OGRE se centra en proporcionar exclusivamente un motor de despliegue gráfico 3D interactivo. OGRE no proporciona mecanismos para gestionr la entrada del usuario. En este módulo utilizaremos OIS (Object Oriented Input System), una biblioteca desarrollada en C++ multiplataforma que permite trabajar con teclado, ratón, joysticks y otros dispositivos de juego. Instalaremos igualmente el paquete binario y las cabeceras.

```
apt-get install libois-1.3.0 libois-dev
```

Tanto OGRE como OIS puede ser igualmente instalado en cualquier distribución compilando directamente los fuentes. En el caso de OGRE, es necesario *CMake* para generar el *makefile* específico para el sistema donde va a ser instalado. En el caso de OIS, es necesario *autotools*.

Microsoft Windows

Aunque no utilizaremos ningún entorno de desarrollo en esta plataforma, dada su amplia comunidad de usuarios, puede ser conveniente generar los ejecutables para esta plataforma. A continuación se detallarán los pasos necesarios para instalar y compilar los desarrollos realizados con OGRE en plataformas Windows.

Como entorno de compilación utilizaremos *MinGW* (*Minimalist GNU for Windows*), que contiene las herramientas básicas de compilación de GNU para Windows.

El instalador de *MinGW* `mingw-get-inst` puede obtenerse de la página web `http://www.mingw.org/`. Puedes instalar las herramientas en `C:\MinGW\`. Una vez instaladas, deberás añadir al *path* el directorio `C:\MinGW\bin`. Para ello, podrás usar la siguiente orden en un terminal del sistema.

```
path = %PATH%;C:\MinGW\bin
```

De igual modo, hay que descargar el SDK de DirectX[4]. Este paso es opcional siempre que no queramos ejecutar ningún ejemplo de OGRE que utilice Direct3D.

Sobre Boost...

Boost es un conjunto de bibliotecas libres que añaden multitud de funcionalidad a la biblioteca de C++ (están en proceso de aceptación por el comité de estandarización del lenguaje).

A continuación instalaremos la biblioteca OGRE3D para MinGW[5]. Cuando acabe, al igual que hicimos con el directorio bin de MinGW, hay que añadir los directorios `boost_1_44\lib\` y `bin\Release` al path.

```
path = %PATH%;C:\Ogre3D\boost_1_44\lib\; C:\Ogre3D\bin\
```

8.7. Hola Mundo en OGRE

A continuación examinaremos un ejemplo básico de funcionamiento de OGRE. Utilizaremos la estructura de directorios mostrada en la Figura 8.19 en los ejemplos desarrollados a lo largo de este módulo.

- En el directorio `include` se incluirán los archivos de cabecera. En este primer ejemplo, no es necesario ningún archivo de cabecera adicional, por lo que este directorio estará vacío.
- El directorio `media` contendrá los archivos de geometría, animaciones y texturas necesarios para ejecutar el ejemplo. Todos estos archivos serán cargados por el *ResourceManager*.
- En `obj` se generarán automáticamente los ficheros objeto compilados a partir de los fuentes existentes en `src`.
- El diretorio `plugins` contendrá los plugins de Ogre. En GNU/Linux, podemos crear un enlace simbólico de este directorio al lugar donde se encuentran los plugins (archivos `.so`) en disco. En Windows deberemos copiar los `.dll` a este directorio. La ruta de este directorio, como veremos más adelante, se debe indicar en el archivo `plugins.cfg`.

[4] Puede descargarse de: http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=6812
[5] Puede descargarse de: http://www.ogre3d.org/download/sdk

- El directorio `src` contiene los ficheros fuente de la aplicación. Gracias al `makefile` que veremos a continuación, la compilación de todos los ficheros fuente en objetos binarios se realiza automáticamente.

Para compilar el ejemplo, definiremos un `makefile` para ambos sistemas operativos. En el siguiente listado se muestra el resultado para GNU/Linux. El formato y uso de *make* será estudiado en detalle a lo largo del primer módulo del curso.

En la línea (0) se define el nombre del ejecutable que queremos obtener. A continuación en las líneas (1-3) se definen los directorios para los archivos fuente, objetos y cabeceras. Las líneas (8) y (11) definen los flags para la compilación y enlazado respectivamente.

El *makefile* construido permite indicar el modo de compilación, utilizando unos flags de compilación en modo *Debug* y otros en modo *Release*. Si llamamos a make con `mode=release`, se utilizarán los flags de compilación optimizada. En otro caso, utilizaremos la compilación con símbolos para el depurado posterior con *GDB*.

En las líneas (23-24) se utilizan las funciones de *make* para generar la lista de objetos a partir de los fuentes `.cpp` existentes en el directorio apuntado por `DIRSRC`. De este modo, se obtienen los objetos con el mismo nombre que los fuentes, pero situados en el directorio indicado por `DIROBJ`.

Finalmente, en las líneas (36-42) se emplean las reglas de compilación implícitas de *make* para generar el ejecutable. Se incluye al final la típica regla de `clean` para limpiar los temporales obtenidos.

De este modo, para compilar el ejemplo en modo *release* ejecutaremos en un terminal

```
make mode=release
```

Si no indicamos modo, o indicamos explícitamente *mode=debug*, se utilizarán los flags de compilación en modo debug.

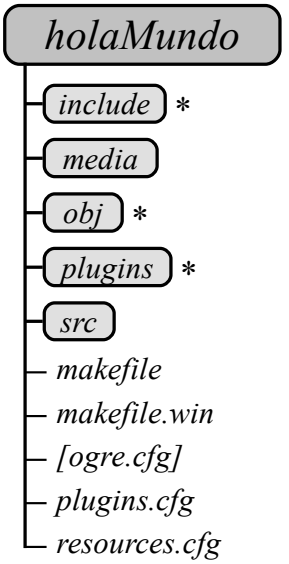


Figura 8.19: Descripción de directorios del ejemplo "*Hola Mundo*".

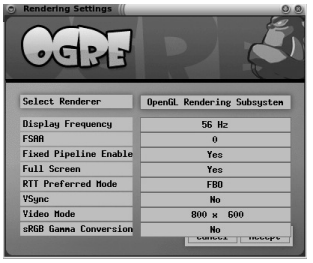


Figura 8.20: Ventana de configuración de las propiedades de *Rendering*.

Listado 8.1: Makefile genérico para GNU/Linux

```
1  EXEC := helloWorld
2  DIRSRC := src/
3  DIROBJ := obj/
4  DIRHEA := include/
5
6  CXX := g++
7
8  # Flags de compilacion ------------------------------
9  CXXFLAGS := -I $(DIRHEA) -Wall `pkg-config --cflags OGRE`
10
11 # Flags del linker ---------------------------------
12 LDFLAGS := `pkg-config --libs-only-L OGRE`
13 LDLIBS := `pkg-config --libs-only-l OGRE` -lOIS -lGL -lstdc++
14
15 # Modo de compilacion (-mode=release -mode=debug) -----------
16 ifeq ($(mode), release)
17     CXXFLAGS += -O2 -D_RELEASE
18 else
19     CXXFLAGS += -g -D_DEBUG
20     mode := debug
21 endif
22
23 # Obtencion automatica de la lista de objetos a compilar -------
24 OBJS := $(subst $(DIRSRC), $(DIROBJ), \
25     $(patsubst %.cpp, %.o, $(wildcard $(DIRSRC)*.cpp)))
26
27 .PHONY: all clean
```

```
28
29 all: info $(EXEC)
30
31 info:
32     @echo '-----------------------------------------------------'
33     @echo '>>> Using mode $(mode)'
34     @echo '    (Please, call "make" with [mode=debug|release])'
35     @echo '-----------------------------------------------------'
36
37 # Enlazado ------------------------------------
38 $(EXEC): $(OBJS)
39     $(CXX) $(LDFLAGS) -o $@ $^ $(LDLIBS)
40
41 # Compilacion ---------------------------------
42 $(DIROBJ)%.o: $(DIRSRC)%.cpp
43     $(CXX) $(CXXFLAGS) -c $< -o $@ $(LDLIBS)
44
45 # Limpieza de temporales ----------------------
46 clean:
47     rm -f *.log $(EXEC) *~ $(DIROBJ)* $(DIRSRC)*~ $(DIRHEA)*~
```


El *makefile* en su versión para plataformas windows se ha nombrado como *makefile.win*. Básicamente es similar al estudiado para GNU/Linux, pero cambian los flags de compilación. Para compilar, ejecutamos el make de *MinGW* que se denomina `mingw32make`:

```
mingw32-make -f makefile-windows mode=release
```

Como se muestra en la Figura 8.19, además de los archivos para make, en el directorio raiz se encuentran tres archivos de configuración. Estudiaremos a continuación su contenido:

- **ogre.cfg**. Este archivo, si existe, intentará ser cargado por el objeto *Root*. Si el archivo no existe, o está creado de forma incorrecta, se le mostrará al usuario una ventana para configurar los parámetros (resolución, método de anti-aliasing, profundidad de color, etc...). En el ejemplo que vamos a realizar, se fuerza a que siempre se abra el diálogo (ver Figura 8.20), recuperando los parámetros que el usuario especificó en la última ejecución.
- **plugins.cfg**. Como hemos comentado anteriormente, un *plugin* es cualquier módulo que implementa alguno de los interfaces de *plugins* de Ogre (como *SceneManager* o *RenderSystem*). En este archivo se indican los plugins que debe cargar OGRE, así como su localización. En el ejemplo del holaMundo, el contenido del archivo indica la ruta al lugar donde se encuentran los plugins (`PluginFolder`), así como el Plugin que emplearemos (pueden especificarse varios en una lista) para el rendering con OpenGL.
- **resources.cfg**. En esta primera versión del archivo de recursos, se especifica el directorio general desde donde OGRE deberá cargar los recursos asociados a los nodos de la escena. A lo largo del curso veremos cómo especificar manualmente más propiedades a este archivo.

Plugins.cfg en Windows

En sistemas Windows, como no es posible crear enlaces simbólicos, se deberían copiar los archivos `.dll` al directorio plugins del proyecto e indicar la ruta relativa a ese directorio en el archivo `plugins.cfg`. Esta aproximación también puede realizarse en GNU/Linux copiando los archivos `.so` a dicho directorio.

Nombres en entidades

Si no indicamos ningún nombre, OGRE elegirá automáticamente un nombre para la misma. El nombre debe ser único. Si el nombre existe, OGRE devolverá una excepción.

Una vez definida la estructura de directorios y los archivos que forman el ejemplo, estudiaremos la docena de líneas de código que tiene nuestro *Hola Mundo*.

Listado 8.2: El main.cpp del Hola Mundo en OGRE

```
1 #include <ExampleApplication.h>
2
3 class SimpleExample : public ExampleApplication {
```

```
4   public : void createScene() {
5     Ogre::Entity *ent = mSceneMgr->createEntity("Sinbad", "Sinbad.
          mesh");
6     mSceneMgr->getRootSceneNode()->attachObject(ent);
7   }
8 };
9
10 int main(void) {
11   SimpleExample example;
12   example.go();
13
14   return 0;
15 }
```

Como vemos, en el main se crea una instancia de la clase *SimpleExample* (definida en las líneas 2-7). Esta clase es derivada de la clase base *ExampleApplication*, que se proporciona en el SDK de OGRE para facilitar el aprendizaje de la biblioteca.

En esta clase definimos el método público *CreateScene*. En la línea 4 llamamos al *SceneManager* (creado automáticamente por la clase base *ExampleApplication*) a través de su puntero, solicitando la creación de una entidad con nombre único `Sinbad`, asociado a `Sinbad.mesh` (que se encontrará en el directorio especificado en `resources.cfg`). Esta llamada nos creará la entidad, y nos devolverá un puntero a un objeto de ese tipo.

A continuación, adjuntamos la entidad creada a la escena. Para ello, accedemos al método *attachObject()* del nodo raíz devuelto por *getRootSceneNode()*.

Si compilamos y ejecutamos el *holaMundo*, primero nos aparecerá una ventana para configurar las opciones de rendering (como se muestra en la Figura 8.20). Cuando pulsemos Accept, se abrirá una ventana con el modelo cargado (muy pequeñito). Si pulsamos la flecha del cursor ↑, el modelo se acercará. Podemos variar la posición de la cámara y la rotación del modelo (así como su modo de rendering) con los otros tres cursores ← ↓ → , y las teclas A, S, D, W y R. La salida del modelo se muestra en la Figura 8.21.

Figura 8.21: Resultado, tras ajustar el modelo, del Hello World.

La clase *ExampleApplication* nos abstrae de la complejidad de crear una aplicación desde cero con OGRE. Utilizaremos esta clase base en algunos ejemplos más del siguiente capítulo, pero la abandonaremos rápidamente para controlar todos los aspectos relativos a la inicialización (carga de Plugins, definición del *RenderSystem*, gestión de la cámara virtual, control de eventos, etc).

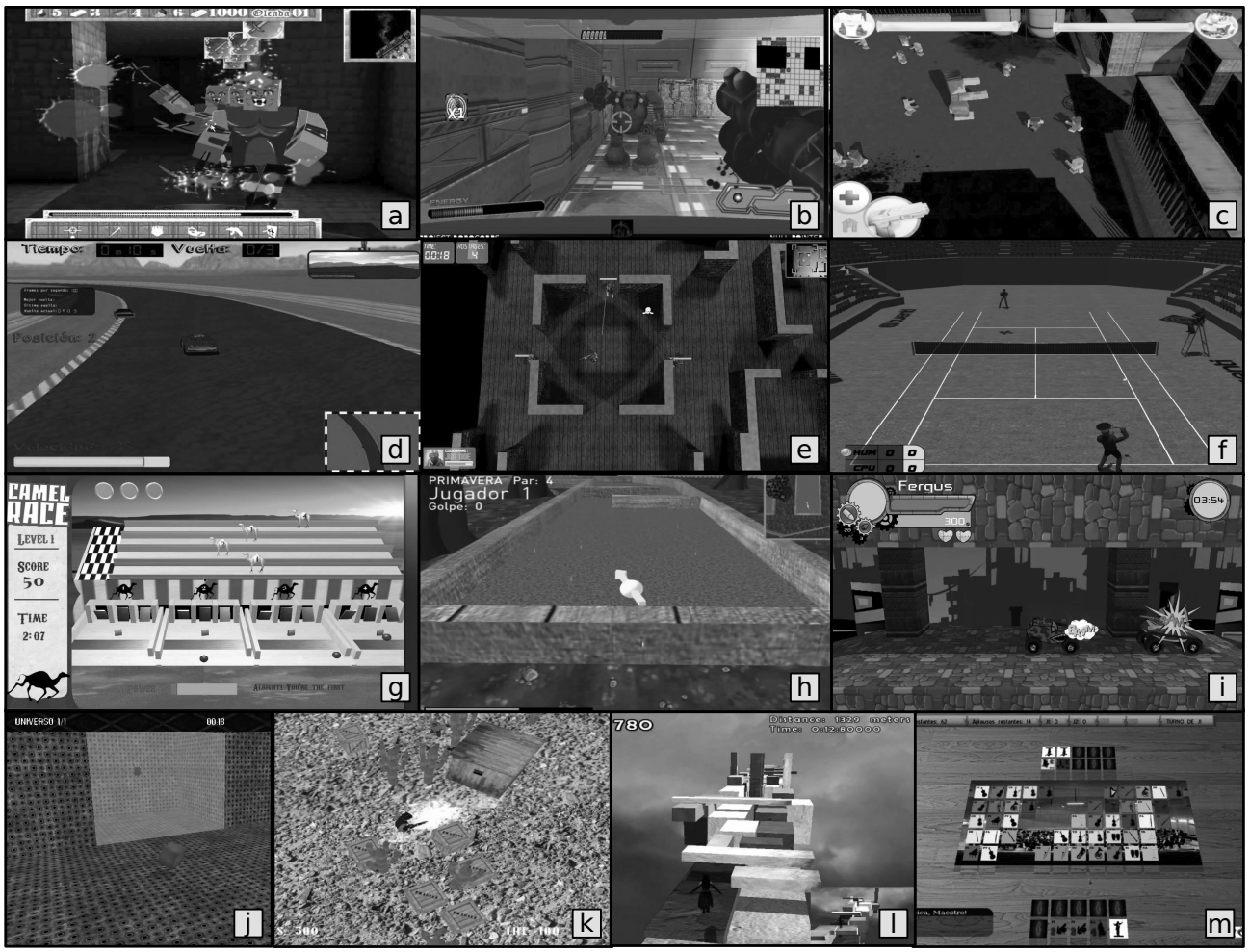


Figura 8.22: Trabajos finales de los alumnos del *Curso de Experto en Desarrollo de Videojuegos* en las Ediciones 2011/2012 y 2012/2013, que utilizaban OGRE como motor de representación gráfica. **a)** Vault Defense *(D. Frutos, J. López, D. Palomares)*, **b)** Robocorps *(E. Aguilar, J. Alcázar, R. Pozuelo, M. Romero)*, **c)** Shadow City *(L.M. García-Muñoz, D. Martín-Lara, E. Monroy)*, **d)** Kartoon *(R. García-Reina, E. Prieto, J.M. Sánchez)*, **e)** Ransom *(M. Blanco, D. Moreno)*, **f)** Crazy Tennis *(J. Angulo)*, **g)** Camel Race *(F.J. Cortés)*, **h)** 4 Season Mini Golf *(A. Blanco, J. Solana)*, **i)** Blood & Metal *(J. García, F. Gaspar, F. Triviño)*, **j)** God Mode On *(J.M. Romero)*, **k)** Another Far West *(R.C. Díaz, J.M. García, A. Suarez)*, **l)** Impulse *(I. Cuesta, S. Fernández)* y **m)** Música Maestro *(P. Santos, C. de la Torre)*.

Capítulo 9

Matemáticas para Videojuegos

Carlos González Morcillo

En este capítulo comenzaremos a estudiar las transformaciones afines básicas que serán necesarias para en el desarrollo de Videojuegos. Las transformaciones son herramientas imprescindibles para cualquier aplicación que manipule geometría o, en general, objetos descritos en el espacio 3D. La mayoría de APIs y motores gráficos que trabajan con gráficos 3D (como OGRE) implementan clases auxiliares para facilitar el trabajo con estos tipos de datos.

9.1. Puntos, Vectores y Coordenadas

Figura 9.1: Aunque en desarrollo de videojuegos se hace uso de prácticamente todas las áreas de las matemáticas (desde trigonometría, álgrebra, estadística o cálculo), en este capítulo nos centraremos en los fundamentos más básicos del álgebra lineal.

Los videojuegos necesitan posicionar objetos en el espacio 3D. El motor gráfico debe gestionar por tanto la posición, orientación y escala de estos objetos (y sus cambios a lo largo del tiempo). Como vimos en el capítulo 8, en gráficos interactivos suelen emplearse representaciones poligionales basadas en triángulos para mejorar la eficiencia. Los vértices de estos triángulos se representan mediante puntos, y las normales mediante vectores. Veamos a continuación algunos conceptos básicos relacionados con los puntos y los vectores.

9.1.1. Sistemas de Referencia

En el inicio del libro *Practical Linear Algebra*, Farin y Hansford [32] reproducen un bonito cuento sobre el pueblo de Schilda, perteneciente a Brandeburgo, que ilustra a la perfección la necesidad de distinguir entre *Sistemas de Referencia*.

En el siglo XVII, un ejército se aproximaba al municipio de Schilda para intentar conquistarlo. El tesoro del pueblo tenía que esconderse para protegerlo de los invasores, por lo que el consejo asesor del ayuntamiento decidió hundirlo en el lago cercano al pueblo. Así, el equipo del ayuntamiento se montaron en un barco y navegaron hasta la mitad del lago. Cuando llegaron, el tesorero del ayuntamiento sacó una navaja de su bolsillo e hizo una muesca profunda en el borde del barco y dejó caer en esa posición el saco con el tesoro. *¿Por qué has hecho eso?* - preguntaron el resto de miembros del equipo. *Porque así podremos saber dónde hemos hundido el tesoro* respondió el audaz tesorero. Todos quedaron impresionados de la astucia del tesorero del pueblo.

Cuando finalizó la guerra, el equipo del ayuntamiento volvió al lago a recuperar el tesoro. Al subir al barco, el elaborado plan del tesorero dejó de ser tan brillante porque no importaba dónde se desplazaran con el barco porque la muesca en el borde de la embarcación ¡marcaba en todos los sitios la posición del tesoro!.

En el siglo XVII *René Descartes* inventó la teoría de los *sistemas de coordenadas*. El tesoro del barco estaba especificado con respecto a su sistema de coordenadas *local*. Sin embargo, es necesario conocer la posición *global* del barco (relativa al origen del lago) para realizar una interpretación correcta de la posición del tesoro.

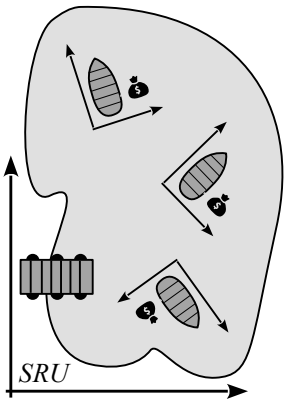


Figura 9.2: Sistemas de referencia Global (Sistema de Referencia Universal o SRU) y Local. Si describimos la posición del tesoro con respecto del sistema de coordenadas local del barco, el tesoro siempre se moverá con el barco y será imposible volver a localizarlo.

9.1.2. Puntos

Un *punto* puede definirse como una localización en un espacio *n*-dimensional. Para identificar claramente p como un punto, decimos que $p \in \mathbb{E}^2$ que significa que un punto 2D vive en el espacio euclídeo $\mathbb{E}^2$. En el caso de los videojuegos, este espacio suele ser bidimensional o tridimensional. El tratamiento discreto de los valores asociados a la posición de los puntos en el espacio exige elegir el tamaño mínimo y máximo que tendrán los objetos en nuestro videojuego. Es crítico definir el convenio empleado relativo al sistema de coordenadas y las unidades entre programadores y artistas. Existen diferentes sistemas de coordenadas en los que pueden especificarse estas posiciones, como se puede ver en la Figura 9.4:

- **Coordenadas Cartesianas**. Es sin duda el sistema de coordenadas más habitual. Este sistema de coordenadas define ejes perpediculares para especificar la posición del punto en el espacio. Como se muestra en la figura 9.3, existen dos convenios para la definición de los ejes de coordenadas cartesianas; según la regla de la mano derecha o de la mano izquierda. Es muy sencillo convertir entre ambos convenios; basta con invertir el valor del eje que cambia.
- **Coordenadas Cilíndricas**. En este sistema se utiliza un eje vertical para definir la altura h, un eje radial r que mide la distancia con ese eje h, y un ángulo de rotación θ definido en la circunferencia sobre ese radio.
- **Coordenadas Esféricas**. Este sistema emplea dos ángulos ϕ y θ, y una distancia radial r.

Coordenadas en OGRE. OGRE utiliza el convenio de la **mano derecha**. El pulgar (eje positivo X) y el índice (eje positivo Y) definen el plano de la pantalla. El pulgar apunta hacia la derecha de la pantalla, y el índice hacia el "techo". El anular define el eje positivo de las Z, saliendo de la pantalla hacia el observador.

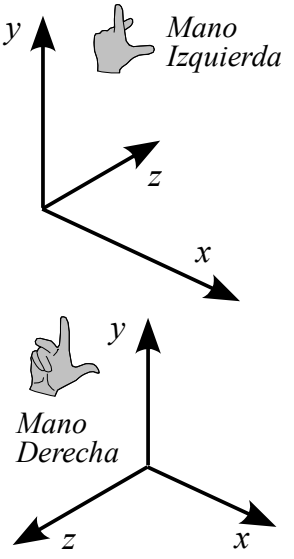


Figura 9.3: Convenios para establecer los sistemas de coordenadas cartesianas.

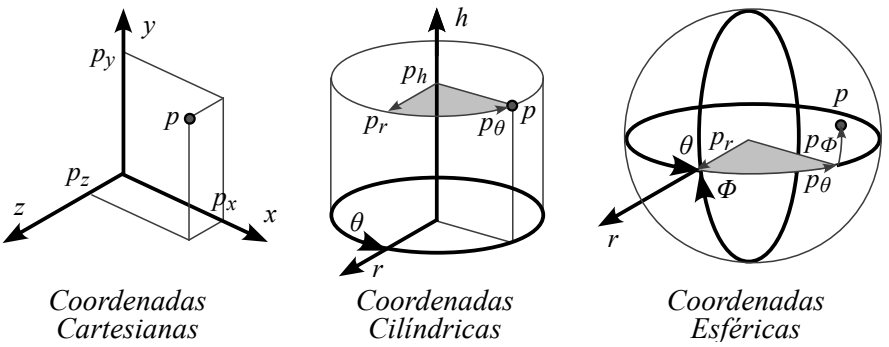


Figura 9.4: Representación de un punto empleando diferentes sistemas de coordenadas.

9.1.3. Vectores

Un vector es una tupla n-dimensional que tiene una longitud (denominada *módulo*), una dirección y un sentido. Puede ser representado mediante una *flecha*. Dos vectores son iguales si tienen la misma longitud, dirección y sentido. Los vectores son entidades *libres* que no están ancladas a ninguna posición en el espacio. Para diferenciar un vector v bidimensional de un punto p, decimos que $v \in \mathbb{R}^2$; es decir, que "vive" en un espacio lineal $\mathbb{R}^2$.

Convenio en OGRE

En OGRE se utilizan las clases `Vector2` y `Vector3` para trabajar indistintamente con puntos y vectores en 2 y 3 dimensiones.

Como en algún momento es necesario representar los vectores como tuplas de números en nuestros programas, en muchas bibliotecas se emplea la misma clase `Vector` para representar puntos y vectores en el espacio. Es imprescindible que el programador distinga en cada momento con qué tipo de entidad está trabajando.

A continuación describiremos brevemente algunas de las operaciones habituales realizadas con vectores.

Suma y resta de vectores

La suma de vectores se puede realizar gráficamente empleando la *regla del paralelogramo*, de modo de la suma puede calcularse como el resultado de unir el inicio del primero con el final del segundo (situando el segundo a continuación del primero). La suma, al igual que con números reales es conmutativa. El vector suma gráficamente "completa el triángulo" (ver Figura 9.5).

Dado un vector a, el vector $-a$ tiene el mismo módulo y dirección que a, pero sentido contrario.

La resta de un vector con otro puede igualmente representarse mediante un paralelogramo. En el diagrama de la Figura 9.5, puede verse cómo efectivamente la resta del vector $b - a$ puede verse como la suma de $(b - a) + a$, como hemos indicado anteriormente.

El resultado de multiplicar un vector a por un valor escalar obtiene como resultado un vector con la misma dirección y sentido, pero con el módulo *escalado* al factor por el que se ha multiplicado. Gráficamente, la operación de multiplicación puede verse como el efecto de *estirar* del vector manteniendo su dirección y sentido.

Las operaciones de suma y resta, multiplicación por un escalar e inversión, se realizan componente a componente del vector:

$$\begin{aligned} a + b &= [(a_x + b_x), (a_y + b_y), (a_z + b_z)] \\ a - b &= [(a_x - b_x), (a_y - b_y), (a_z - b_z)] \\ n \cdot a &= (n \cdot a_x, n \cdot a_y, n \cdot a_z) \\ -a &= (-a_x, -a_y, -a_z) \end{aligned}$$

Recordemos que, la suma de dos vectores da como resultado un vector, mientras que la suma de un punto y un vector se interpreta como la obtención del *punto destino* de aplicar la transformación del vector sobre el punto. La resta de dos puntos ($p_b - p_a$) da como resultado un vector, con módulo igual a la distancia existente entre ambos puntos, la dirección de la recta que pasa por ambos puntos y el sentido de ir del punto p_a a p_b.

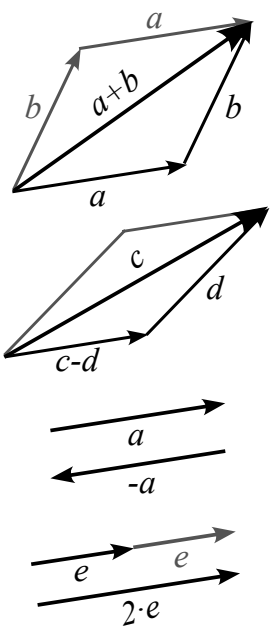


Figura 9.5: Representación de algunas operaciones con vectores. Comenzando por arriba: suma y resta de vectores, inversión y multiplicación por escalar.

Módulo y normalización

El módulo de un vector es un valor escalar que representa la longitud del mismo. Puede calcularse como:

$$|a| = \sqrt{a_x^2 + a_y^2 + a_z^2}$$

Si dividimos un vector por su longitud (es decir, escalamos el vector a por el factor $1/|a|$), obtenemos como resultado un vector con la misma dirección y sentido, pero de módulo la unidad (vector *unitario*). Esta operación se denomina *normalización*, y da como resultado un vector *normalizado* que se emplea en gran cantidad de algoritmos en programación de videojuegos.

$$a_N = \frac{a}{|a|}$$

Los vectores pueden multiplicarse entre sí, pero a diferencia de los escalares, admiten dos operaciones de multiplicación; el *producto escalar* (que da como resultado un escalar), y el *producto vectorial* (que lógicamente da como resultado otro vector). Veamos a continuación cómo se definen estas dos operaciones y algunas de sus aplicaciones prácticas.

Producto Escalar

El *producto escalar* (*dot product*) es conmutativo ($a \cdot b = b \cdot a$), y se calcula como:

$$a \cdot b = a_x b_x + a_y b_y + a_z b_z \tag{9.1}$$

También puede ser calculado mediante la siguiente expresión que relaciona el ángulo θ que forman ambos vectores.

$$a \cdot b = |a|\,|b|\,cos(\theta) \tag{9.2}$$

Combinando las expresiones 9.1 y 9.2, puede calcularse el ángulo que forman dos vectores (operación muy utilizada en informática gráfica).

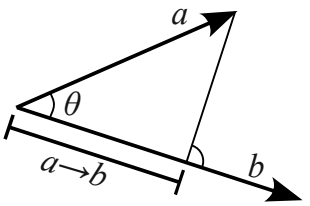


Figura 9.6: La proyección de a sobre b obtiene como resultado un escalar.

Otra operación muy útil es el cálculo de la *proyección* de un vector sobre otro, que se define $a \rightarrow b$ como la longitud del vector a que se proyecta mediante un ángulo recto sobre el vector b (ver Figura 9.6).

$$a \rightarrow b = |a|\, cos(\theta) = \frac{a \cdot b}{|b|}$$

El producto escalar se emplea además para detectar si dos vectores tienen la misma dirección, o si son perpendiculares, o si tienen la misma dirección o direcciones opuestas (para, por ejemplo, eliminar geometría que no debe ser representada).

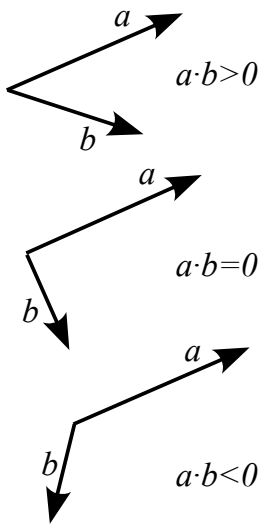


Figura 9.7: Algunos ejemplos de uso del producto escalar.

- **Colineal**. Dos vectores son colineales si se encuentran definidos en la misma línea recta. Si *normalizamos* ambos vectores, y aplicamos el producto escalar podemos saber si son *colineales* con el mismo sentido ($a \cdot b = 1$) o sentido opuesto ($a \cdot b = -1$).
- **Perpendicular**. Dos vectores son perpendiculares si el ángulo que forman entre ellos es 90 (o 270), por lo que $a \cdot b = 0$.
- ***Misma* dirección**. Si el ángulo que forman entre ambos vectores está entre 270 y 90 grados, por lo que $a \cdot b > 0$ (ver Figura 9.7).
- **Dirección *Opuesta***. Si el ángulo que forman entre ambos vectores es mayor que 90 grados y menor que 270, por lo que $a \cdot b < 0$.

Esta interpretación de *misma* dirección y dirección *opuesta* no es literal. Nos servirá para comprobar si el vector normal de una cara poligonal está *de frente* a la cámara virtual (si tiene dirección opuesta al vector *look*) de la cámara, o por el contrario está siendo visto *de espaldas*.

Producto Vectorial

Mediante el *producto vectorial* (*cross product*) de dos vectores se obtiene otro vector que es perpendicular a los dos vectores originales.

$$a \times b = [(a_y b_z - a_z b_y), (a_z b_x - a_x b_z), (a_x b_y - a_y b_x)]$$

El sentido del vector resultado del producto escalar sigue la regla de la *mano derecha* (ver Figura 9.8): si *agarramos* $a \times b$ con la palma de la mano, el pulgar apunta en el sentido positivo del vector producto si el giro del resto de los dedos va de a a b. En caso contrario, el sentido del vector es el inverso. Por tanto, el producto vectorial no es conmutativo.

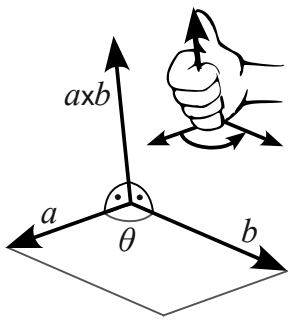


Figura 9.8: Representación del produto vectorial. El módulo del producto vectorial $a \times b$ es igual al área del paralelogramo representado en la figura.

El módulo del producto vectorial está directamente relacionado con el ángulo que forman ambos vectores θ. Además, el módulo del producto vectorial de dos vectores es igual al área del paralelogramo formado por ambos vectores (ver Figura 9.8).

$$|a \times b| = |a|\, |b|\, sin\theta$$

El producto vectorial se utiliza en gran cantidad de situaciones. El cálculo del vector perpendicular a dos vectores es útil para calcular el vector normal asociado a un triángulo (habitualmente se devuelve normalizado para su posterior utilización en la etapa de *shading*).

Más Mates?, Síií!! Existen multitud de objetos matemáticos empleados en gráficos interactivos, como rayos, planos, segmentos, así como estructuras de datos para la aceleración de los cálculos. Estos elementos serán estudiados en cada sección que haga uso de ellos.

A continuación describiremos las transformaciones geométricas más comunes empleadas en gráficos por computador. Para introducir los conceptos generales asociados a las transformaciones, comenzaremos con una discusión sobre las operaciones en 2D para pasar a la notación matricial 2D y, posteriormente, a la generalización tridimensional empleando coordenadas homogeneas.

9.2. Transformaciones Geométricas

En la representación de gráficos 3D es necesario contar con herramientas para la transformación de los objetos básicos que compondrán la escena. En gráficos interactivos, estas primitivas son habitualmente conjuntos de triángulos que definen mallas poligonales. Las operaciones que se aplican a estos triángulos para cambiar su posición, orientación y tamaño se denominan **transformaciones geométricas**. En general podemos decir que una transformación toma como entrada elementos como vértices y vectores y los convierte de *alguna manera*.

La transformación básica bidimensional más sencilla es la **traslación**. Se realiza la traslación de un punto mediante la suma de un vector de desplazamiento a las coordenadas iniciales del punto, para obtener una nueva posición de coordenadas. Si aplicamos esta traslación a *todos* los puntos del objeto, estaríamos desplazando ese objeto de una posición a otra. De este modo, podemos definir la traslación como la suma de un vector libre de traslación t a un punto original p para obtener el punto trasladado p' (ver Figura 9.9). Podemos expresar la operación anterior como:

$$p'_x = p_x + t_x \qquad p'_y = p_y + t_y \tag{9.3}$$

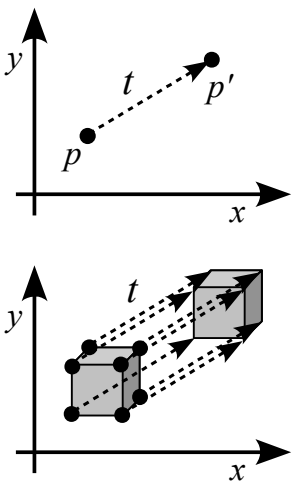


Figura 9.9: Arriba. Traslación de un punto p a p' empleando el vector t. **Abajo**. Es posible trasladar un objeto poligonal completo aplicando la traslación a todos sus vértices.

De igual modo podemos expresar una **rotación** de un punto $p = (x, y)$ a una nueva posición rotando un ángulo θ respecto del origen de coordenadas, especificando el eje de rotación y un ángulo θ. Las coordenadas iniciales del punto se pueden expresar como (ver Figura 9.10):

$$p_x = d\,cos\alpha \qquad p_y = d\,sen\alpha \tag{9.4}$$

Siendo d la distancia entre el punto y el origen del sistema de coordenadas. Así, usando identidades trigonométricas se pueden expresar las coordenadas transformadas como la suma de los ángulos del punto original α y el que queremos rotar θ como:

$$p'_x = d\,cos(\alpha + \theta) = d\,cos\alpha\,cos\theta - d\,sen\alpha\,sen\theta$$
$$p'_y = d\,sen(\alpha + \theta) = d\,cos\alpha\,sen\theta - d\,sen\alpha\,cos\theta$$

Que sustituyendo en la ecuación 9.4, obtenemos:

$$p'_x = p_x\,cos\theta - p_y\,sen\theta \qquad p'_y = p_x\,sin\theta - p_y\,cos\theta \tag{9.5}$$

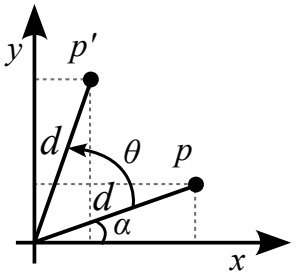


Figura 9.10: Rotación del punto p un ángulo θ respecto del origen de coordenadas.

De forma similar, un **cambio de escala** de un objeto bidimensional puede llevarse a cabo multiplicando las componentes x, y del objeto por el factor de escala S_x, S_y en cada eje. Así, como se muestra en la Figura 9.11 un cambio de escala se puede expresar como:

$$p'_x = p_x S_x \qquad p'_y = p_y S_y \tag{9.6}$$

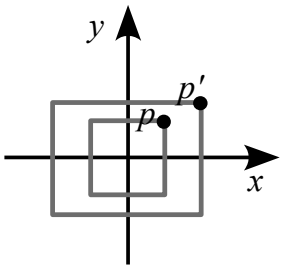


Figura 9.11: Conversión de un cuadrado a un rectángulo empleando los factores de escala $S_x = 2, S_y = 1{,}5$.

Cuando queremos cambiar la localización de un objeto, habitualmente necesitamos especificar una **combinación de traslaciones y rotaciones** en el mismo (por ejemplo, cuando cogemos el teléfono móvil de encima de la mesa y nos lo guardamos en el bolsillo, sobre el objeto se aplican varias traslaciones y rotaciones). Es interesante por tanto disponer de alguna representación que nos permita combinar transformaciones de una forma eficiente.

C9

9.2.1. Representación Matricial

En muchas aplicaciones gráficas los objetos deben transformarse geométricamente de forma constante (por ejemplo, en el caso de una animación, en la que en cada *frame* el objeto debe cambiar de posición. En el ámbito de los videojuegos, es habitual que aunque un objeto permanezca inmóvil, es necesario cambiar la posición de la cámara virtual para que se ajuste a la interacción con el jugador.

Homogeneízate!!

Gracias al uso de coordenadas homogéneas es posible representar las ecuaciones de transformación geométricas como multiplicación de matrices, que es el método estándar en gráficos por computador (soportado en hardware por las tarjetas aceleradoras gráficas).

De este modo resulta crítica la eficiencia en la realización de estas transformaciones. Como hemos visto en la sección anterior, las ecuaciones 9.3, 9.5 y 9.6 nos describían las operaciones de traslación, rotación y escalado. Para la primera es necesario realizar una *suma*, mientras que las dos últimas requieren *multiplicaciones*. Sería conveniente poder **combinar las transformaciones** de forma que la posición final de las coordenadas de cada punto se obtenga de forma directa a partir de las coordenadas iniciales.

Si reformulamos la escritura de estas ecuaciones para que todas las operaciones se realicen multiplicando, podríamos conseguir homogeneizar estas transformaciones.

Si añadimos un término extra (parámetro homogéneo h) a la representación del punto en el espacio (x, y), obtendremos la **representación homogénea** de la posición descrita como (x_h, y_h, h). Este *parámetro homogéneo* h es un valor distinto de cero tal que $x = x_h/h$, $y = y_h/h$. Existen, por tanto infinitas representaciones homogéneas equivalentes de cada par de coordenadas, aunque se utiliza normalmente $h = 1$. Como veremos en la sección 9.3, no siempre el parámetro h es igual a uno.

Puntos y Vectores

En el caso de puntos, la componente homogénea $h = 1$. Los vectores emplean como parámetro $h = 0$.

De este modo, la operación de traslación, que hemos visto anteriormente, puede expresarse de forma matricial como:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{9.7}$$

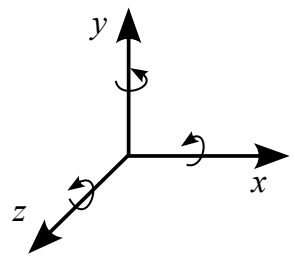


Figura 9.12: Sentido de las rotaciones positivas respecto de cada eje de coordenadas.

Al resolver la multiplicación matricial se obtienen un conjunto de ecuaciones equivalentes a las enunciadas en 9.3. De forma análoga, las operaciones de rotación T_r y escalado T_s tienen su equivalente matricial homogéneo.

$$T_r = \begin{bmatrix} cos\theta & -sen\theta & 0 \\ sen\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad T_s = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{9.8}$$

Las transformaciones inversas pueden realizarse sencillamente cambiando el signo en el caso de la traslación y rotación (distancias y ángulos negativos), y en el caso de la escala, utilizando los valores $1/S_x$ y $1/S_y$.

Las **transformaciones en el espacio 3D** requieren simplemente añadir el parámetro homogéneo y describir las matrices (en este caso 4x4). Así, las traslaciones T_t y escalados T_s en 3D pueden representarse de forma homogénea mediante las siguientes matrices:

$$T_t = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T_s = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9.9}$$

Las rotaciones requieren distinguir el eje sobre el que se realizará la rotación. Las **rotaciones positivas** alrededor de un eje se realizan en sentido opuesto a las agujas del reloj, cuando se está mirando a lo largo de la mitad positiva del eje hacia el origen del sistema de coordenadas (ver Figura 9.12).

Las expresiones matriciales de las rotaciones son las siguientes:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sen\theta & 0 \\ 0 & sen\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_y = \begin{bmatrix} cos\theta & 0 & sen\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sen\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_z = \begin{bmatrix} cos\theta & -sen\theta & 0 & 0 \\ sen\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9.10}$$

Las tres transformaciones estudiadas (traslación, rotación y escalado) son ejemplos de **transformaciones afines**, en las que cada una de las coordenadas transformadas se pueden expresar como una función lineal de la posición origen, y una serie de constantes determinadas por el tipo de transformación.

Una subclase de las *transformaciones afines*, es la formada por las transformaciones lineales, que permiten aplicar todas las transformaciones mediante multiplicaciones. Como hemos visto anteriormente, las traslación afines no homogénea no es una operación lineal (porque no puede realizarse mediante una multiplicación). Las transformaciones afines mantienen la mayoría de las propiedades de los objetos, excepto los ángulos y las distancias. Aplicando transformaciones afines se mantiene la *colineridad* (ver Tabla 9.1), por lo que las líneas paralelas seguirán siendo paralelas.

Como caso particular de estudio, las *transformaciones de cuerpo rígido* preservan todas las propiedades geométricas de los objetos. Cualquier combinación de *rotaciones y traslaciones homogéneas* son transformaciones de cuerpo rígido.

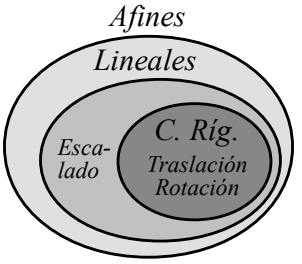


Figura 9.13: Esquema de subclases de transformaciones afines y operaciones asociadas.

Cuadro 9.1: Propiedades geométricas preservadas según la clase de transformación: Transformaciones afines, Lineales y de Cuerpo Rígido.

Propiedad	T. Afines	T. Lineales	T. Cuerpo Rígido
Ángulos	No	No	Sí
Distancias	No	No	Sí
Ratios de distancias	Sí	Sí	Sí
Líneas paralelas	Sí	Sí	Sí
Líneas rectas	Sí	Sí	Sí

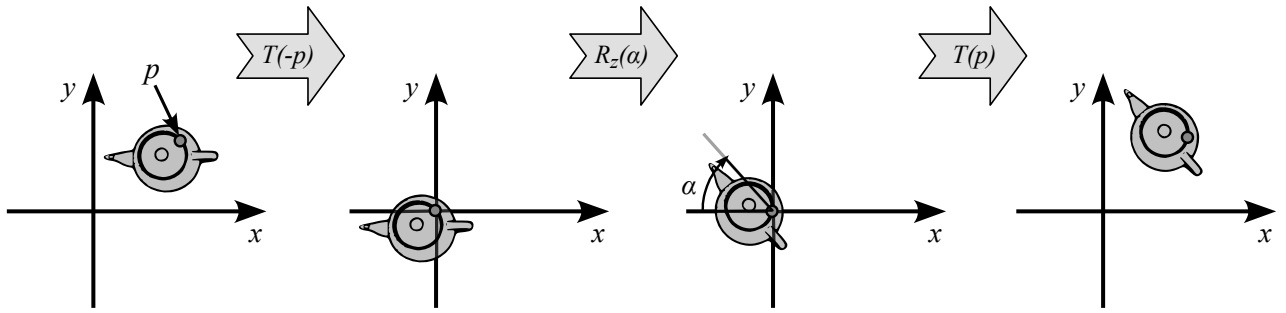


Figura 9.14: Secuencia de operaciones necesarias para rotar una figura respecto de un origen arbitrario.

9.2.2. Transformaciones Inversas

En muchas situaciones resulta interesante calcular la inversa de una matriz. Un ejemplo típico es en la resolución de ecuaciones, como en el caso de la expresión $A = Bx$. Si queremos obtener el valor de B, tendríamos $B = A/x$. Por desgracia, las matrices no tienen asociado un operador de división, por lo que debemos usar el concepto de *matriz inversa*.

Para una matriz A, se define su inversa A^{-1} como la matriz que, multiplicada por A da como resultado la matriz identidad I:

$$A \cdot A^{-1} = A^{-1} \cdot A = I \tag{9.11}$$

Matrices Inversas

No todas las matrices tienen inversa (incluso siendo cuadradas). Un caso muy simple es una matriz cuadrada cuyos elementos son cero.

Tanto la matriz A como su inversa deben ser *cuadradas* y del mismo tamaño. Otra propiedad interesante de la inversa es que la inversa de la inversa de una matriz es igual a la matriz original $(A^{-1})^{-1} = A$.

En la ecuación inicial, podemos resolver el sistema utilizando la matriz inversa. Si partimos de $A = B \cdot x$, podemos multiplicar ambos términos a la izquierda por la inversa de B, teniendo $B^{-1} \cdot A = B^{-1} \cdot B \cdot x$, de forma que obtenemos la matriz identidad $B^{-1} \cdot A = I \cdot x$, con el resultado final de $B^{-1} \cdot A = x$.

En algunos casos el cálculo de la matriz inversa es directo, y puede obtenerse de forma intuitiva. Por ejemplo, en el caso de una traslación pura (ver ecuación 9.9), basta con emplear como factor de traslación el mismo valor en negativo. En el caso de escalado, como hemos visto bastará con utilizar $1/S$ como factor de escala.

Cuando se trabaja con matrices compuestas, el cálculo de la inversa tiene que realizarse con métodos generales, como por ejemplo el método de eliminación de Gauss o la traspuesta de la matriz adjunta.

9.2.3. Composición

Como hemos visto en la sección anterior, una de las principales ventajas derivadas del trabajo con sistemas homogéneos es la **composición de matrices**. Matemáticamente esta *composición* se realiza multiplicando las matrices en un orden determinado, de forma que es posible obtener la denominada **matriz de transformación neta** M_N resultante de realizar sucesivas transformaciones a los puntos. De este modo, bastará con multiplicar la M_N a cada punto del modelo para obtener directamente su posición final. Por ejemplo, si P es el punto original y P' es el punto transformado, y $T_1 \cdots T_n$ son transformaciones (rotaciones, escalados, traslaciones) que se aplican al

punto P, podemos expresar la transformación neta como:

$$P' = T_n \times \cdots \times T_2 \times T_1 \times P$$

Este orden de multiplicación de matrices es el habitual empleado en gráficos por computador, donde las transformaciones se premultiplican (la primera está más cerca del punto original P, más a la derecha.

La matriz de transformación neta M_N se definiría como $M_N = T_n \times \cdots T_2 \times T_1$, de tal forma que sólo habría que calcularla una vez para todos los puntos del modelo y aplicarla a todos vértices en su posición original para obtener su posición final. De este modo, si un objeto poligonal está formado por V vértices, habrá que calcular la matriz de transformación neta M_N y aplicarla una vez a cada vértice del modelo.

$$P' = M_N \times P$$

Otro aspecto a tener en cuenta es que la expresión de las transformaciones para trabajar con coordenadas homogeneas, que se han comentado en las ecuaciones 9.9 y 9.10 se refieren al **Sistema de Referencia Universal** (*SRU*) o Sistema de Referencia Global.

Conmutatividad. Recordemos que la multiplicación de matrices es asociativa, pero *no* es conmutativa, por lo que el orden de aplicación de las transformaciones es importante (ver Figura 9.17). La propiedad asociativa se utiliza para *resumir* en la *Matriz de Transformación Neta* M_N la secuencia de transformaciones que se aplicarán sobre los modelos. De este modo, bastará con multiplicar una única matriz a todos los vértices.

De este modo, si se quiere realizar una transformación respecto de un punto distinto a ese origen del SRU, habrá que hacer coincidir primero el punto con el origen del sistema de referencia, aplicar la transformación y devolver el objeto a su posición original. Así, en el ejemplo de la Figura 9.14 si queremos rotar el objeto respecto del punto *p* es necesario, primero trasladar el objeto para que su origen quede alineado con el origen del *SRU*, luego aplicar la rotación, y finalmente aplicar la traslación inversa. Así, tenemos $M_N = T_{-p} \times R_z \times T_p$.

$$\begin{bmatrix} RS_{11} & RS_{12} & RS_{13} & P_x \\ RS_{21} & RS_{22} & RS_{23} & P_y \\ RS_{31} & RS_{32} & RS_{33} & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figura 9.15: El formato de la matriz de transformación neta permite identificar la posición final del objeto (traslación) en la cuarta columna $P_x P_y P_z$. La matriz 3×3 interior combina las rotaciones y escalados que se aplicarán al objeto.

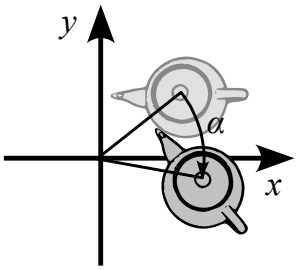


Figura 9.16: Resultado de aplicar directamente la rotación $R_z(\alpha)$ respecto del SRU. Puede comprobarse el diferente resultado obtenido en comparación con el de la figura 9.14.

9.3. Perspectiva: Representación Matricial

Como vimos en la sección 8.2.4, en la proyección en perspectiva se tomaban como entrada las coordenadas de visualización, generando la proyección de los objetos sobre el plano de imagen.

Vimos en la Figura 8.9 que, empleando triángulos semejantes, obteníamos las siguientes coordenadas:

$$\frac{p'_x}{p_x} = \frac{-d}{p_z} \quad \Leftrightarrow \quad p'_x = \frac{-d\, p_x}{p_z} \tag{9.12}$$

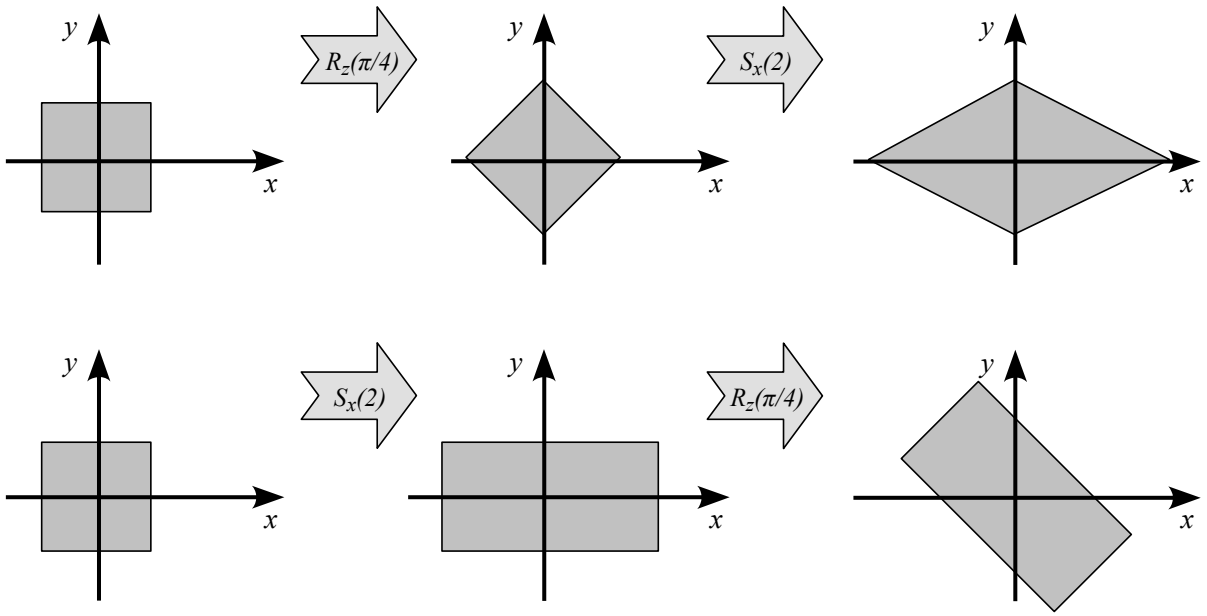


Figura 9.17: La multiplicación de matrices no es conmutativa, por lo que el orden de aplicación de las transformaciones es relevante para el resultado final. Por ejemplo, la figura de arriba primero aplica una rotación y luego el escalado, mientras que la secuencia inferior aplica las transformaciones en orden inverso.

De igual forma obtenemos la coordenada $p'_y = -d\, p_y/p_z$, y $p'_z = -d$. Estas ecuaciones se pueden expresar fácilmente de forma matricial (siendo M_p la matriz de proyección en perspectiva):

$$p' = M_p\, p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ -p_z/d \end{bmatrix} = \begin{bmatrix} -d\, p_x/p_z \\ -d\, p_y/p_z \\ -d \\ 1 \end{bmatrix} \tag{9.13}$$

Definición *near* y *far*

Un error común suele ser que la definición correcta de los planos *near* y *far* únicamente sirve para limitar la geometría que será representada en el *Frustum*. La definición correcta de estos parámetros es imprescindible para evitar errores de precisión en el Z-Buffer.

El último paso de la ecuación 9.13 corresponde con la normalización de los componentes dividiendo por el parámetro homogéneo $h = -p_z/d$. De esta forma tenemos la matriz de proyección que nos *aplasta* los vértices de la geometría sobre el plano de proyección. Desafortunadamente esta operación no puede *deshacerse* (no tiene inversa). La geometría una vez *aplastada* ha perdido la información sobre su componente de profundidad. Es interesante obtener una trasnformación en perspectiva que proyecte los vértices sobre el *cubo unitario* descrito previamente (y que sí puede deshacerse).

De esta forma, definimos la *pirámide de visualización* o *frustum*, como la pirámide truncada por un plano paralelo a la base que define los objetos de la escena que serán representados. Esta *pirámide de visualización* queda definida por cuatro vértices que definen el plano de proyección (left l, right r, top t y bottom b), y dos distancias a los planos de recorte (near n y far f), como se representa en la Figura 9.18. El *ángulo de visión* de la cámara viene determinado por el ángulo que forman l y r (en horizontal) y entre t y b (en vertical).

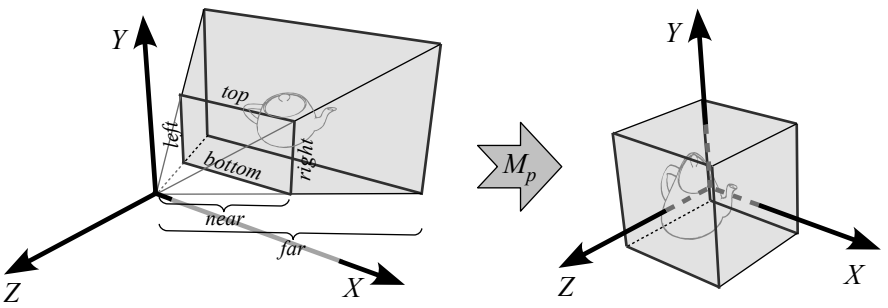


Figura 9.18: La matriz M_p se encarga de transformar la *pirámide de visualización* en el cubo unitario.

La matriz que transforma el *frustum* en el cubo unitario viene dada por la expresión de la ecuación 9.14.

$$M_p = \begin{bmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{9.14}$$

Si se usa este tipo de proyección, el valor de profundidad normalizado no cambia linealmente con la entrada, sino que se va perdiendo precisión. Es recomendable situar los planos de recorte cercano y lejano (distancias n y f en la matriz) lo más juntos posibles para evitar errores de precisión en el *Z-Buffer* en distancias grandes.

9.4. Cuaternios

Los cuaternios (*quaternion*) fueron propuestos en 1843 por William R. Hamilton como extensión de los números complejos. Los cuaternios se representan mediante una 4-tupla $q = [q_x, q_y, q_z, q_w]$. El término q_w puede verse como un término **escalar** que se añade a los tres términos que definen un **vector** (q_x, q_y, q_z). Así, es común representar el cuaternio como $q = [q_v, q_s]$ siendo $q_v = (q_x, q_y, q_z)$ y $q_s = q_w$ en la tupla de cuatro elementos inicial.

Los cuaternios unitarios, que cumplen la restricción de que $(q_x^2+q_y^2+q_z^2+q_w^2) = 1$, se emplean ampliamente para representar rotaciones en el espacio 3D. El conjunto de todos los cuaternios unitarios definen la hiperesfera unitaria en $\mathbb{R}^4$.

Si definimos como a al vector unitario que define el eje de rotación, θ como el ángulo de rotación sobre ese eje empleando la regla de *la mano derecha* (si el pulgar apunta en la dirección de a, las rotaciones positivas se definen siguiendo la dirección de los dedos curvados de la mano), podemos describir el cuaternio como (ver Figura 9.19):

$$q = [q_v, q_s] = [a\, sin(\theta/2),\, cos(\theta/2)]$$

De forma que *la parte vectorial* se calcula escalando el vector de dirección unitario por el seno de $\theta/2$, y la parte escalar como el coseno de $\theta/2$. Obviamente, esta multiplicación en la parte vectorial se realiza componente a componente.

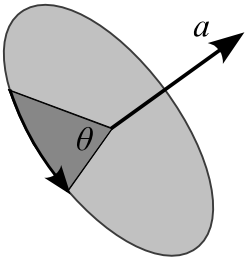


Figura 9.19: Representación de un cuaternio unitario.

Teorema de Euler

Según el Teorema de Rotación de Leonhard Euler (1776), cualquier composición de rotaciones sobre un sólido rígido es equivalente a una sola rotación sobre un eje, llamado *Polo de Euler*.

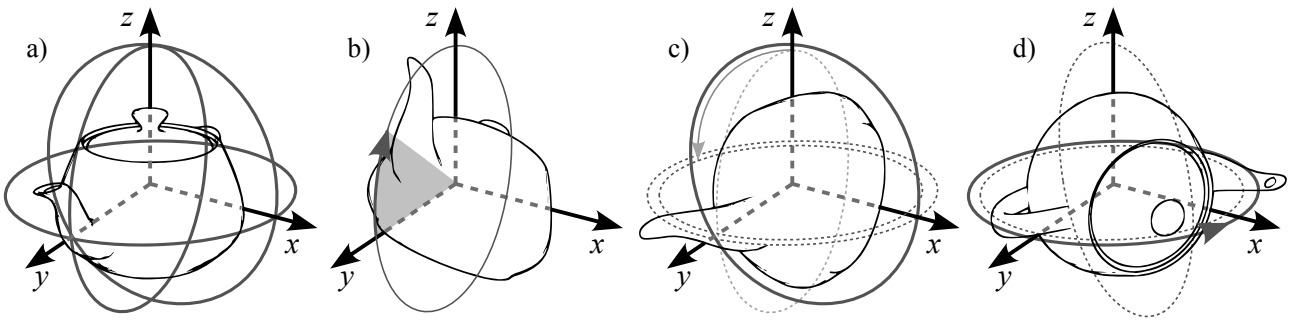


Figura 9.20: El problema del bloqueo de ejes (*Gimbal Lock)*. En **a)** se muestra el convenio de sistema de coordenadas que utilizaremos, junto con las elipses de rotación asociadas a cada eje. En **b)** aplicamos una rotación respecto de x, y se muestra el resultado en el objeto. El resultado de aplicar una rotación de 90º en y se muestra en **c)**. En esta nueva posición cabe destacar que, por el orden elegido de aplicación de las rotaciones, ahora el eje x y el eje z están alineados, perdiendo un grado de libertad en la rotación. En **d)** se muestra cómo una rotación en z tiene el mismo efecto que una rotación en x, debido al efecto de *Gimbal Lock*.

Como vimos en la sección 9.2.1, se puede usar una matriz para *resumir* cualquier conjunto de rotaciones en 3D. Sin embargo, las matrices no son la mejor forma de representar rotaciones, debido a:

1. **Almacenamiento**. La representación mediante matrices require nueve valores de punto flotante para almacenar una rotación. Según el *Teorema de Rotación de Euler* es suficiente con establecer la rotación frente a un único eje (el Polo de Euler).

2. **Tiempo de Cómputo**. En la composición de la matriz con un vector, se necesitan calcular nueve multiplicaciones y seis sumas en punto flotante.

3. **Interpolación**. En muchas ocasiones es necesario calcular valores intermedios entre dos puntos conocidos. En el caso de rotaciones, es necesario calcular los valores de rotación intermedios entre dos rotaciones clave (en la animación de la rotación de una cámara virtual, o en las articulaciones de un personaje animado). La interpolación empleando matrices es muy complicada.

4. **Bloque de Ejes**. El problema del bloqueo de ejes (denominado *Gimbal Lock*) ocurre cuando se trabaja con ángulos de Euler. Como hemos visto, cualquier rotación se puede descomponer en una secuencia de tres rotaciones básicas sobre cada uno de los ejes. Como hemos visto, el orden de aplicación de las rotaciones importa (no es conmutativo), por lo que tendremos que decidir uno. Por ejemplo, podemos aplicar primero la rotación en x, a continuación en y y finalmente en z. El orden de las rotaciones es relevante porque rotaciones posteriores tienen influencia jerárquica sobre las siguientes. El problema de *Gimbal Lock* ocurre cuando, por accidente uno de los ejes queda alineado con otro, reduciendo los grados de libertad del objeto. En el ejemplo de la Figura 9.20, tras aplicar la rotación de 90º sobre el eje y en *c)*, el eje x sufre la misma transformación (por arrastrar la rotación en y por jerarquía), de modo que queda alineado con Z. Ahora, la rotación en z equivale a la rotación en x, por lo que hemos perdido un grado de libertad.

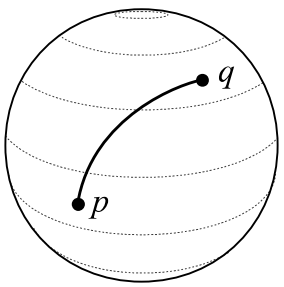


Figura 9.21: Los cuaternios unitarios pueden ser representados como puntos sobre la esfera unitaria. La interpolación esférica lineal entre dos cuaternios p y q se puede igualmente representar como un arco sobre la esfera..

De este modo, es interesante trabajar con cuaternios para la especificación de rotaciones. Veamos a continuación algunas de las operaciones más comunmente utilizadas con esta potente herramienta matemática.

9.4.1. Suma y Multiplicación

La suma de dos cuaternios se realiza sumando la parte escalar y vectorial por separado:

$$p + q = [(p_v + q_v), (p_s + q_s)]$$

La multiplicación de dos cuaternios representa su composición (es decir, el resultado de aplicar la rotación de uno a continuación del otro). Existen diferentes formas de multiplicar cuaternios. A continuación definiremos la más ampliamente utilizada en gráficos por computador, que es la multiplicación de *Grassman*:

$$pq = [(p_s q_v + q_s p_v + p_v \times q_v), (p_s q_s - p_v \cdot q_v)]$$

Como vemos, este tipo de multiplicación utiliza el producto vectorial y la suma de vectores en la parte vectorial de cuaternio, y el producto escalar en la parte escalar del cuaternio.

9.4.2. Inversa

Inversión de cuaternios

La inversa de un cuaternio es mucho más rápida que el cálculo de la inversa de una matriz cuadrada. Ésta es otra razón más por la que elegir cuaternios para especificar rotaciones.

En el caso de cuaternios unitarios, la inversa del cuaternio q^{-1} es igual al conjugado q^*. El conjugado se calcula simplemente negando la parte vectorial, por lo que, trabajando con cuaternios unitarios, podemos calcularlos simplemente como:

$$q^{-1} = q^* = [-q_v, q_s]$$

La multiplicación de un cuaternio por su inversa obtiene como resultado el escalar 1 (es decir, rotación 0). De este modo, $q\, q^{-1} = [0\, 0\, 0\, 1]$.

9.4.3. Rotación empleando Cuaternios

Para aplicar una rotación empleando cuaternios, primero convertimos el punto o el vector que rotaremos a cuaternio. En ambos casos, se le aplica 0 como término de la parte escalar. Así, dado el vector v, el cuaternio correspondiente a v se calcula como $v = [v\, 0] = [v_x\, v_y\, v_z\, 0]$.

Una vez que tenemos expresado en formato de cuaternio el vector, para aplicar la rotación empleando el cuaternio, primero lo pre-multiplicamos por q y posteriormente lo post-multiplicamos por el inverso (o el conjugado, en el caso de trabajar con cuaternios unitarios). Así, el resultado de la rotación v' puede expresarse como:

Asociatividad

La concatenación de cuaternios cumple la propiedad asociativa, por lo que es posible obtener la expresión del *cuaternio neto* y aplicarlo a varios objetos de la escena.

$$v' = qvq^{-1}$$

La concatenación de cuaternios funciona exactamente igual que la concatenación de matrices. Basta con multiplicar los cuaternios entre sí. Si quisiéramos aplicar las rotacioens de los cuaternios q_1 y q_2 sobre el vector v (especificado en formato de cuaternio), bastará con realizar la operación:

$$v' = q_2 q_1\, v\, q_1^{-1} q_2^{-1}$$

Es habitual realizar conversiones entre matrices y cuaternios. OGRE incorpora llamadas a su biblioteca (ver sección 9.6) para realizar esta operación. Se recomienda el artículo de Gamasutra de Nick Bobic *Rotating Objects Using Quaternions* para profundizar en su uso.

9.5. Interpolación Lineal y Esférica

Una de las operaciones más empleadas en gráficos por computador es la interpolación. Un claro ejemplo de su uso es para calcular la posición intermedia en una animación de un objeto que se traslada desde un punto A, hasta un punto B.

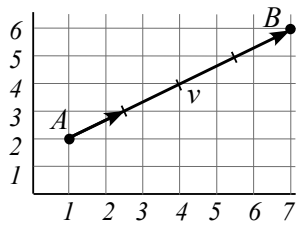


Figura 9.22: Ejemplo de interpolación lineal entre dos puntos. Se han marcado sobre el vector $\vec{v}$ los puntos correspondientes a $t = 0,25$, $t = 0{,}5$ y $t = 0{,}75$.

La *interpolación lineal* es un método para calcular valores intermedios mediante polinomios lineales. Es una de las formas más simples de interpolación. Habitualmente se emplea el acrónimo *LERP* para referirse a este tipo de interpolación. Para obtener valores intermedios entre dos puntos A y B, habitualmente se calcula el vector $\vec{v} = B - A$ con origen en A y destino en B.

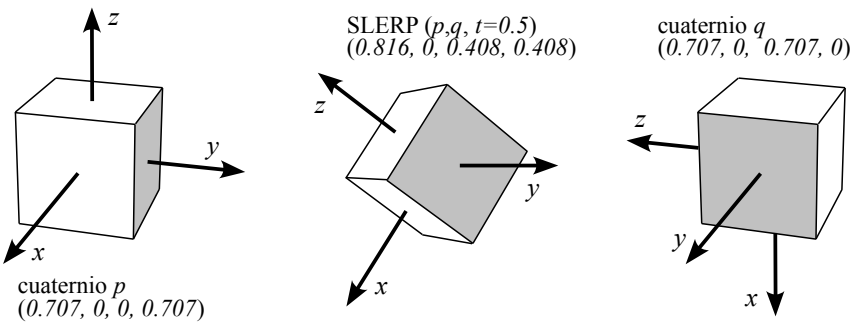


Figura 9.23: Ejemplo de aplicación de interpolación esférica *SLERP* en la rotación de dos cuaternios p y q sobre un objeto.

Para calcular valores intermedios, se utiliza una representación paramétrica, de forma que, modificando el valor de $t \in (0, 1)$, obtenemos puntos intermedios:

$$LERP(A, B, u) = A + \vec{v}t$$

Como podemos ver en el ejemplo de la Figura 9.22, para $t = 0{,}25$ tenemos que $\vec{v} = (6, 4)$, por lo que $\vec{v} \cdot 0{,}25 = (1{,}5, 1)$. Así, $LERP(A, B, 0{,}25) = (1, 2) + (1{,}5, 1) = (2{,}5, 3)$.

Como se enunció en la sección 9.4, una de las ventajas del uso de cuaternios es la relativa a la facilidad de realizar interpolación entre ellos. La *Interpolación Lineal Esférica* (también llamada *slerp* de *Spherical Linear Interpolation*) es una operación que, dados dos cuaternios y un parámetro $t \in (0, 1)$, calcula un cuaternio interpolado entre ambos, mediante la siguiente expresión[1]:

$$SLERP(p, q, t) = \frac{sin((1-t)\theta)}{sin(\theta)}p + \frac{sin(t\theta)}{sin(\theta)}q$$

[1] Siendo θ el ángulo que forman los dos cuaternios unitarios.

9.6. El Módulo Math en OGRE

Documentación...

Recordemos que en distribuciones de GNU/Linux basadas en Debian, puedes encontrar la documentación de OGRE en local en `/usr/share/doc/ogre-doc`.

La biblioteca de clases de OGRE proporciona una amplia gama de funciones matemáticas para trabajar con matrices, vectores, cuaternios y otras entidades matemáticas como planos, rayos, esferas, etc...

La documentación relativa a todas estas clases se encuentra en el *Módulo Math*, que forma parte del núcleo (*Core*) de OGRE. A continuación se muestra un sencillo ejemplo que utiliza algunos operadores de la clase `Vector3` y `Quaternion`.

Listado 9.1: Ejemplo de algunas clases de Math

```
 1 cout << " Ejemplo de algunas clases de Math en OGRE " << endl;
 2 cout << "-----------------------------------------" << endl;
 3
 4 Vector3 v1(1.0, 0.0, 0.0);
 5 Vector3 v2(0.0, 2.0, 0.0);
 6 Quaternion p(0.707107, 0.0, 0.0, 0.707107);
 7 Quaternion q(Degree(90), Vector3(0.0, 1.0, 0.0));
 8
 9 cout << " Vector V1 = " << v1 << endl;
10 cout << " Vector V2 = " << v2 << endl;
11 cout << " Cuaternio P = " << p << endl;
12 cout << " Cuaternio Q = " << q << endl;
13
14 cout << "--- Algunos operadores de Vectores ------" << endl;
15 cout << " Suma: V1 + V2 = " << v1 + v2 << endl;
16 cout << " Producto por escalar: V1 * 7.0 = " << v1*7.0 << endl;
17 cout << " P. escalar: V1 * V2 = " << v1.dotProduct(v2) << endl;
18 cout << " P. vectorial: V1 x V2 =" << v1.crossProduct(v2) << endl;
19 cout << " Modulo: |V1| = " << v1.length() << endl;
20 cout << " Normalizar: V2n = " << v2.normalisedCopy() << endl;
21 cout << " Angulo (V1,V2)= " << v1.angleBetween(v2).valueDegrees()
       << endl;
22 cout << "--- Algunos operadores de Cuaternios ----" << endl;
23 cout << " Suma: P + Q = " << p + q << endl;
24 cout << " Producto: P * Q = " << p * q << endl;
25 cout << " Producto escalar: P * Q = " << p.Dot(q) << endl;
26 cout << " SLERP(p,q,0.5)= "<< Quaternion::Slerp(0.5, p, q) << endl;
```

Ogre facilita la creación de estos objetos proporcionando diversos constructores. Por ejemplo, en la línea [7] se emplea un constructor de cuaternio que permite especificar por separado el ángulo de rotación y el vector.

Operadores de alto nivel

Como puede verse en el listado anterior, OGRE incorpora multitud de operadores de alto nivel para sumar, restar, multiplicar y volcar por pantalla los objetos de la clase `Vector` y `Quaternion`.

Algunas clases, como la clase *Quaternion* incorpora funciones auxiliares como el método de interpolación lineal esférico SLERP (spherical linear interpolation). En la línea [26] se emplea para obtener el cuaternio interpolado para $t = 0{,}5$. La Figura 9.23 muestra el resultado de este caso de aplicación concreto. El cuaternio definido en p se corresponde con una rotación de 90º en Z (del SRU (Sistema de Referencia Universal)), mientras que el cuaternio definido en q equivale a una rotación de 90º en Y (del SRU). A continuación se muestra el resultado de ejecutar el código anterior:

```
 Ejemplo de algunas clases de Math en OGRE
-----------------------------------------
 Vector V1 = Vector3(1, 0, 0)
 Vector V2 = Vector3(0, 2, 0)
 Cuaternio P = Quaternion(0.707107, 0, 0, 0.707107)
 Cuaternio Q = Quaternion(0.707107, 0, 0.707107, 0)
--- Algunos operadores de Vectores ------
 Suma: V1 + V2 = Vector3(1, 2, 0)
 Producto por escalar: V1 * 7.0 = Vector3(7, 0, 0)
 Producto escalar: V1 * V2 = 0
 Producto vectorial: V1 x V2 = Vector3(0, 0, 2)
```

```
 Modulo: |V1| = 1
 Normalizar: V2n = Vector3(0, 1, 0)
 Angulo entre V1 y V2 = 90
--- Algunos operadores de Cuaternios ----
 Suma: P + Q = Quaternion(1.41421, 0, 0.707107, 0.707107)
 Producto: P * Q = Quaternion(0.5, -0.5, 0.5, 0.5)
 Producto escalar: P * Q = 0.5
 SLERP(p,q,0.5) = Quaternion(0.816497, 0, 0.408248, 0.408248)
```

9.7. Ejercicios Propuestos

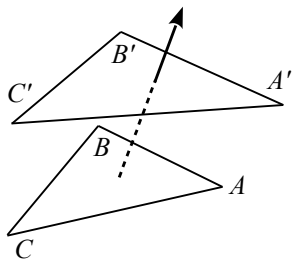


Figura 9.24: En el ejercicio 4, tras aplicar el desplazamiento de 2 unidades en la dirección del vector normal sobre los vértices originales del triángulo, obtenemos $A'B'C'$.

1. Dado el polígono ABC, con A=(1,1,3), B=(3,1,3) y C=(1,1,1), calcular el restultado de aplicar una traslación de -2 unidades en el eje X, y 1 unidad en el eje Z, y después una rotación de $\pi/2$ radianes respecto del eje Z. Las transformaciones se realizan en ese orden y sobre el SRU.

2. Dado el cuadrado definido por los puntos A=(1,0,3), B=(3,0,3), C=(3,0,1) y D=(1,0,1), realizar un escalado del doble en el eje X respecto a su centro geométrico. Compruebe que el resultado no es el mismo que si realiza el escalado directamente con respecto del eje X del SRU.

3. ¿Cuáles serían las expresiones matriciales correspondientes a la operación de reflexión (espejo) sobre los planos ortogonales x=0, y=0 y z=0? Calcule una matriz por cada plano.

4. Dado el triángulo ABC, siendo A=(1, -1, 1), B=(-1, -1, -1) y C=(1, 1, -1), desplace la cara poligonal 2 unidades en la dirección de su vector normal (ver Figura 9.24). Suponga que el vector normal sale según la regla de la mano derecha.

Capítulo 10

Grafos de Escena

Carlos González Morcillo

Como vimos en el capítulo 8, uno de los pilares clave de cualquier motor gráfico es la organización de los elementos que se representarán en la escena. Esta gestión se realiza mediante el denominado *Grafo de Escena* que debe permitir inserciones, búsquedas y métodos de ordenación eficientes. OGRE proporciona una potente aproximación a la gestión de los *Grafos de Escena*. En este capítulo trabajaremos con los aspectos fundamentales de esta estructura de datos jerárquica.

10.1. Justificación

Como señala D. Eberly [30], el motor de despliegue gráfico debe dar soporte a cuatro características fundamentales, que se apoyan en el *Grafo de Escena*:

Test de Visibilidad

La gestión de la visibilidad de los elementos de la escena es una de las tareas típicas que se encarga de realizar el motor gráfico empleando el *Grafo de Escena*.

1. **Gestión Datos Eficiente**. Es necesario eliminar toda la geometría posible antes de enviarla a la GPU. Aunque la definición del *Frustum* y la etapa de *recorte* eliminan la geometría que no es visible, ambas etapas requieren tiempo de procesamiento. Es posible utilizar el conocimiento sobre el tipo de escena a representar para optimizar el envío de estas *entidades* a la GPU. Las relaciones entre los objetos y sus atributos se modelan empleando el *Grafo de Escena*.

2. **Interfaz de Alto Nivel**. El *Grafo de Escena* puede igualmente verse como un interfaz de alto nivel que se encarga de alimentar al motor gráfico de bajo nivel empleando alguna API específica. El diseño de este interfaz de alto nivel permite abstraernos de futuros cambios necesarios en las capas dependientes del dispositivo de visualización.

3. **Facilidad de Uso**. Esta característica está directamente relacionada con la anterior. El *Grafo de Escena* forma parte de la especificación del middleware de despliegue gráfico que facilita el uso al programador final del videojuego. De este modo, el programador se centra en la semántica asociada a la utilización del *Grafo*, dejando de lado los detalles de representación de bajo nivel.

4. **Extensibilidad**. En gran cantidad de videojuegos comerciales, los equipos de trabajo de programación frecuentemente se quejan del cambio en la especificación de requisitos iniciales por parte del equipo artístico. Esto implica añadir soporte a nuevos tipos de geometría, efectos visuales, etc. El motor debe estar preparado para añadir los nuevos tipos sin necesidad de cambios profundos en la arquitectura. El diseño correcto del *Grafo de Escena* mitiga los problemas asociados con la extensibilidad y la adaptación al cambio en los requisitos iniciales.

Las estructuras de datos de *Grafos de Escena* suelen ser grafos que representen las relaciones jerárquicas en el despliegue de objetos compuestos. Esta dependencia espacial se modela empleando *nodos* que representan agregaciones de *elementos* (2D o 3D), así como transformaciones, procesos y otras entidades representables (sonidos, vídeos, etc...).

El **Grafo de Escena** puede definirse como un grafo dirigido sin ciclos. Los arcos del grafo definen dependencias del nodo hijo respecto del padre, de modo que la aplicación de una transformación en un nodo padre hace que se aplique a todos los nodos hijo del grafo.

El *nodo raíz* habitualmente es un nodo abstracto que proporciona un punto de inicio conocido para acceder a la escena. Gracias al grafo, es posible especificar fácilmente el movimiento de escenas complejas de forma relativa a los elementos padre de la jerarquía. En la Figura 10.1, el movimento que se aplique a la *Tierra* (rotación, traslación, escalado) se aplicará a todos los objetos que dependan de ella (como por ejemplo a las *Tiritas*, o las *Llantas* de la exacavadora). A su vez, las modificaciones aplicadas sobre la *Cabina* se aplicarán a todos los nodos que herenden de la jerarquía, pero no al nodo *Llantas* o al nodo *Tierra*.

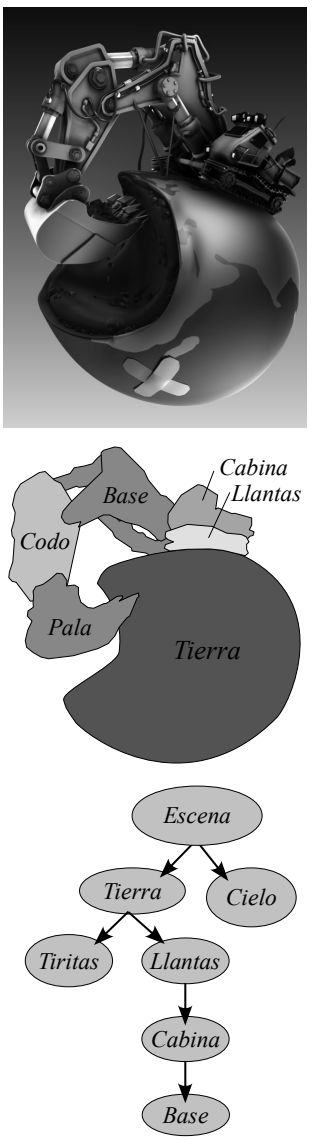


Figura 10.1: Ejemplo de grafo asociado a la escena de la excavadora. Escena diseñada por Manu Järvinen.

10.1.1. Operaciones a Nivel de Nodo

Como señala Theoharis et al. [98], en términos funcionales, la principal ventaja asociada a la construcción de un *Grafo de Escena* es que cada operación se propaga de forma jerárquica al resto de entidades mediante el recorrido del grafo. Las principales operaciones que se realizan en cada nodo son la *inicialización*, *simulación*, *culling* y *dibujado*. A continuación estudiaremos qué realiza cada operación.

- **Inicialización**. Este operador establece los valores iniciales asociados a cada entidad de la jerarquía. Es habitual que existan diferentes *instancias* referenciadas de las entidades asociadas a un nodo. De este modo se separan los datos estáticos asociados a las entidades (definición de la geometría, por ejemplo) y las transformaciones aplicados sobre ellos que se incluyen a nivel de nodo.

- **Simulación**. En esta operación se determinan los parámetros y las variables asociadas a los tipos de datos existentes en el nodo. En el caso de animaciones, se actualizan los controladores asociados para que reflejen el instante de tiempo actual.

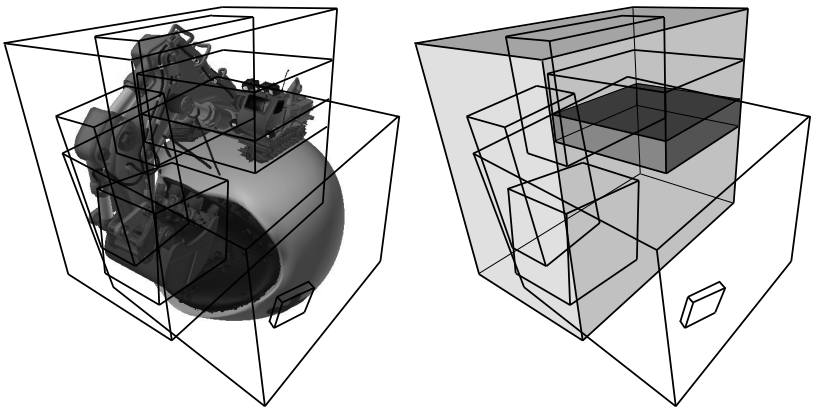


Figura 10.2: Ejemplo de estructura de datos para la gestión del *culling* jerárquico. En el ejemplo, el nodo referente a las *Llantas* (ver Figura 10.1) debe mantener la información de la caja límite *(Bounding Box)* de todos los hijos de la jerarquía, remarcada en la imagen de la derecha.

- **Culling**. En esta operación se estudia la visibilidad de los elementos contenidos en los nodos. Gracias a la especificación jerárquica del grafo, es posible *podar* ramas completas de la jerarquía. Para realizar esta operación, se utilizan estructuras de datos adicionales que mantienen información sobre las coordenadas límite (máximo y mínimo) asociadas a cada objeto. Así, consultando estos límites, si un nodo queda totalmente fuera de la pirámide de visualización (*Frustum*), implicará que todos sus nodos hijo están igualmente fuera y no será necesario su posterior dibujado (ver Figura 10.2).
- **Dibujado**. En esta operación se aplican los algoritmos de *rendering* a cada nodo de la jerarquía (comenzando por la raíz, *bajando* hasta las hojas). Si un nodo contiene órdenes que cambien el modo de dibujado, se aplicarán igualmente a todos los nodos hijo de la jerarquía.

A continuación estudiaremos el interfaz de alto nivel que proporciona OGRE para la gestión de *Grafos de Escena*. Veremos las facilidades de gestión *orientada a objetos*, así como la abstracción relativa a los tipos de datos encapsulados en cada nodo.

Tareas del Gestor

En este capítulo nos centraremos en la parte específica de aplicación de transformaciones a los objetos. A lo largo del módulo estudiaremos otros aspectos relacionados con el gestor, como el tratamiento del *culling* o la gestión de sombras dinámicas.

10.2. El Gestor de Escenas de OGRE

Como hemos visto en la sección anterior, el *Gestor de Escenas*, apoyado en el *Grafo de Escena*, permite optimizar los datos que se representarán finalmente, podando parte de la geometría que forma la escena. El *Gestor de Escenas* en OGRE se encarga de las siguientes tareas:

- Gestión, creación y acceso eficiente a objetos móviles, luces y cámaras.
- Carga y ensamblado de la geometría (estática) del mundo 3D.
- Implementación de la operación de *Culling* para eliminación de superficies no visibles.
- Dibujado de todos los elementos que forman la escena.

- Gestión y representación de sombras dinámicas.
- Interfaz para realizar consultas a la escena. Estas consultas son del tipo: *¿Qué objetos están contenidos en una determinada región del espacio 3D?*

10.2.1. Creación de Objetos

La tarea más ampliamente utilizada del *Gestor de Escenas* es la creación de los objetos que formarán la escena: luces, cámaras, sistemas de partículas, etc. Cualquier elemento que forme parte de la escena será gestionado por el *Gestor de Escenas*, y formará parte del *Grafo de Escena*. Esta gestión está directamente relacionada con el ciclo de vida completa de los objetos, desde su creación hasta su destrucción.

Los **Nodos** del *Grafo de Escena* se crean empleando el *Gestor de Escena*. Los nodos del grafo en OGRE tienen asignado un único nodo padre. Cada nodo padre puede tener cero o más hijos. Los nodos pueden *adjuntarse (attach)* o *separarse (detach)* del grafo en tiempo de ejecución. El nodo no se destruirá hasta que se le indique explícitamente al *Gestor de Escenas*.

En la inicialización, el *Gestor de Escena* se encarga de crear al menos un nodo: el *Root Node*. Este nodo no tiene padre, y es el padre de toda la jerarquía. Aunque es posible aplicar transformaciones a cualquier nodo de la escena (como veremos en la sección 10.2.2), al nodo *Root* no se le suele aplicar ninguna transformación y es un buen punto en el que adjuntar toda la geometría estática de la escena. Dado el objeto `SceneManager`, una llamada al método `getRootSceneNode()` nos devuelve un puntero al **Root SceneNode**. Este nodo es, en realidad, una variable miembro de la clase `SceneManager`.

Los objetos de la escena se pueden adjuntar a cualquier nodo de la misma. En un nodo pueden existir varios objetos. Sin embargo, no es posible adjuntar la misma instancia de un objeto a varios nodos de escena al mismo tiempo. Para realizar esta operación primero hay que *separar (detach)* el objeto del nodo, y posteriormente *adjuntarlo (attach)* a otro nodo distinto.

Para **crear un nodo** en OGRE, se utiliza el método `createSceneNode`, que puede recibir como parámetro el nombre del nodo. Si no se especifica, OGRE internamente le asigna un nombre único que podrá ser accedido posteriormente.

Visibilidad

Si quieres que un nodo no se dibuje, simplemente ejecutas la operación de *detach* y no será *renderizado*.

Destrucción de nodos

La destrucción de un *nodo* de la escena no implica la liberación de la memoria asignada a los objetos adjuntos al nodo. Es responsabilidad del programador liberar la memoria de esos objetos.

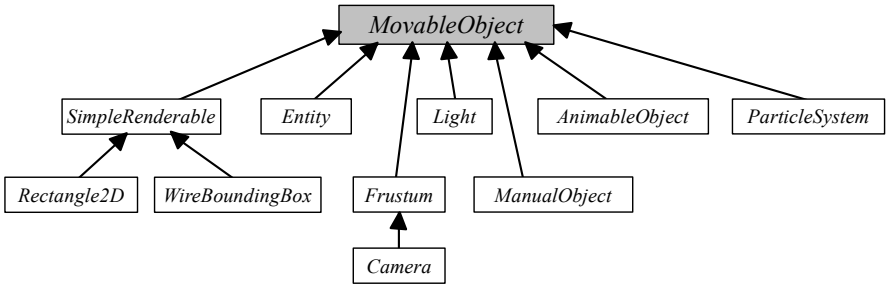


Figura 10.4: Algunos de los tipos de objetos que pueden ser añadidos a un nodo en OGRE. La clase abstracta *MovableObject* define *pequeños* objetos que serán añadidos *(attach)* al nodo de la escena.

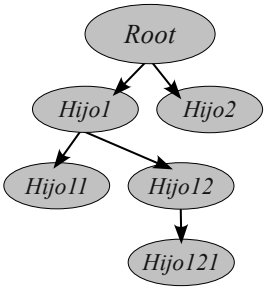


Figura 10.3: Ejemplo de grafo de escena válido en OGRE. Todos los nodos (salvo el *Root*) tienen un nodo padre. Cada nodo padre tiene cero o más hijos.

La operación de **añadir un nodo hijo** se realiza mediante la llamada al método `addChild(Node *child)` que añade el nodo hijo previamente creado al nodo existente. La clase `SceneNode` es una subclase de la clase abstracta `Node`, definida para contener información sobre las transformaciones que se aplicarán a los nodos hijo. De este modo, la transformación neta que se aplica a cada hijo es el resultado de componer las de sus padres con las suyas propias.

Como hemos visto antes, los nodos de la escena contienen objetos. OGRE define una clase abstracta llamada `MovableObject` para definir todos los tipos de objetos que pueden ser añadidos a los nodos de la escena. En la Figura 10.4 se definen algunos de las principales subclases de esta clase abstracta. Para **añadir un objeto** a un nodo de la escena, se emplea la llamada al método `attachObject(MovableObject *obj)`.

La clase `Entity` se emplea para la gestión de pequeños objetos móviles basados en mallas poligonales. Para la definición del escenario (habitualmente con gran complejidad poligonal e inmóvil) se emplea la clase `StaticGeometry`. La **creación de entidades** se realiza empleando el método `createEntity` del Gestor de Escena, indicando el nombre del modelo y el nombre que quiere asignarse a la entidad.

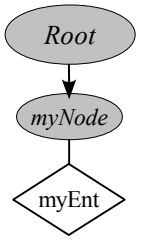


Figura 10.5: Jerarquía obtenida en el grafo de escena asociado al listado de ejemplo.

El código del siguiente listado (modificación del *Hola Mundo* del capítulo 8) crea un nodo hijo `myNode` de *RootSceneNode* en las líneas [3-4], y añade la entidad `myEnt` al nodo `myChild` en la línea [6] (ver Figura 10.5). A partir de ese punto del código, cualquier transformación que se aplique sobre el nodo *myNode* se aplicarán a la entidad *myEnt*.

Listado 10.1: Creación de nodos

```
1 class SimpleExample : public ExampleApplication {
2   public : void createScene() {
3     SceneNode* node = mSceneMgr->createSceneNode("myNode");
4     mSceneMgr->getRootSceneNode()->addChild(node);
5     Entity *myEnt = mSceneMgr->createEntity("cuboejes", "cuboejes.
          mesh");
6     node->attachObject(myEnt);
7   }
8 };
```

10.2.2. Transformaciones 3D

Las transformaciones 3D se realizan a nivel de **nodo** de escena, no a nivel de objetos. En realidad en OGRE, los elementos que se mueven son los *nodos*, no los objetos individuales que han sido añadidos a los nodos.

Como vimos en la sección 9.1, OGRE utiliza el convenio de la mano derecha para especificar su sistema de coordenadas. Las rotaciones positivas se realizan en contra del sentido de giro de las agujas del reloj (ver Figura 9.12).

Las transformaciones en el *Grafo de Escena* de OGRE siguen el convenio general explicado anteriormente en el capítulo, de forma que se especifican de forma relativa al nodo padre.

Traslación

La traslación de un nodo que ya ha sido posicionado anteriormente se realiza mediante la llamada a `translate`.

La **traslación** absoluta se realiza mediante la llamada al método `setPosition`, que admite un *Vector3* o tres argumentos de tipo Real. Esta traslación se realiza de forma relativa al nodo padre de la jerarquía. Para recuperar la traslación relativa de un nodo con respecto a su nodo padre se puede emplear la llamada `getPosition`.

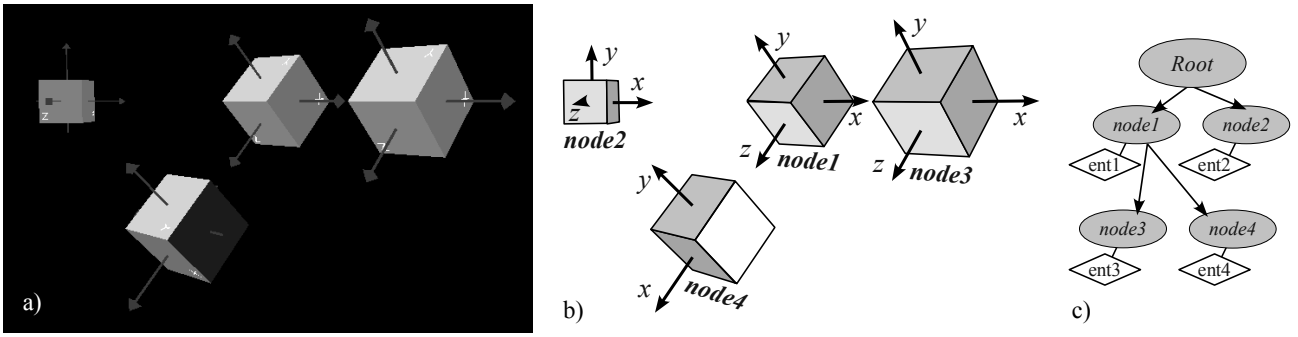


Figura 10.6: Ejemplo de transformaciones empleando el *Grafo de Escena* de OGRE. **a)** Resultado de ejecución del ejemplo. **b)** Representación de los nodos asociados a la escena. **c)** Jerarquía del grafo de escena.

La **rotación** puede realizarse mediante diversas llamadas a métodos. Uno de los más sencillos es empleando las llamadas a `pitch`, `yaw` y `roll` que permiten especificar la rotación (en radianes) respecto del eje X, Y y Z respectivamente. La forma más general de aplicar una rotación es mediante la llamada a `rotate` que require un eje de rotación y un ángulo o un cuaternio como parámetro. De igual modo, la llamada a `getOrientation` nos devuelve la rotación actual del nodo.

El **escalado** análogamente se realiza mediante la llamada al método `setScale`. Mediante el método `getScale` obtenemos el factor de escala aplicado al nodo.

A continuación veremos un código que utiliza algunas de las llamadas a métodos estudiadas anteriormente para aplicar transformaciones en 3D. El resultado de la ejecución de dicho código se muestra en la Figura 10.6.

Listado 10.2: Ejemplo de uso del Grafo de Escena

```
1  class SimpleExample : public ExampleApplication {
2   public : void createScene() {
3    SceneNode* node1 = mSceneMgr->createSceneNode("Node1");
4    Entity *ent1 = mSceneMgr->createEntity("ent1", "cuboejes.mesh");
5    node1->attachObject(ent1);
6    mSceneMgr->getRootSceneNode()->addChild(node1);
7    node1->setPosition(0,0,480);
8    node1->yaw(Degree(-45));
9    node1->pitch(Radian(Math::PI/4.0));
10
11   SceneNode* node2 = mSceneMgr->createSceneNode("Node2");
12   Entity *ent2 = mSceneMgr->createEntity("ent2", "cuboejes.mesh");
13   node2->attachObject(ent2);
14   mSceneMgr->getRootSceneNode()->addChild(node2);
15   node2->setPosition(-10,0,470);
16
17   SceneNode* node3 = mSceneMgr->createSceneNode("Node3");
18   Entity *ent3 = mSceneMgr->createEntity("ent3", "cuboejes.mesh");
19   node3->attachObject(ent3);
20   node1->addChild(node3);
21   node3->setPosition(5,0,0);
22
23   SceneNode* node4 = mSceneMgr->createSceneNode("Node4");
24   Entity *ent4 = mSceneMgr->createEntity("ent4", "cuboejes.mesh");
25   node4->attachObject(ent4);
26   node1->addChild(node4);
27   node4->setPosition(0,0,5);
28   node4->yaw(Degree(-90));
29  }
30 };
```

En el listado anterior se utilizan las funciones de utilidad para convertir a radianes (línea 7), así como algunas constantes matemáticas (como π en la línea 8).

Posición de Root

Aunque es posible aplicar transformaciones al nodo *Root*, se desaconseja su uso. El nodo *Root* debe mantenerse estático, como punto de referencia del SRU.

Como se observa en la Figura 10.6, se han creado 4 nodos. La transformación incial aplicada al *node1* es *heredada* por los nodos 3 y 4. De este modo, basta con aplicar una traslación de 5 unidades en el eje x al nodo 3 para obtener el resultado mostrado en la Figura 10.6.b). Esta traslación es relativa al sistema de referencia *transformado* del nodo padre. No ocurre lo mismo con el nodo 2, cuya posición se especifica de nuevo desde el origen del sistema de referencia universal (del nodo *Root*).

Los métodos de trasnformación de 3D, así como los relativos a la gestión de nodos cuentan con múltiples versiones sobrecargadas. Es conveniente estudiar la documentación de la API de OGRE para emplear la versión más interesante en cada momento.

10.2.3. Espacios de transformación

Las transformaciones estudiadas anteriormente pueden definirse relativas a diferentes espacios de transformación. Muchos de los métodos explicados en la sección anterior admiten un parámetro opcional de tipo `TransformSpace` que indica el espacio de transformación *relativo* de la operación. OGRE define tres espacios de trasnformación como un tipo enumerado `Node::TransformSpace`:

- **TS_LOCAL**. Sistema de coordenadas local del nodo.
- **TS_PARENT**. Sistema de coordenadas del nodo padre.
- **TS_WORLD**. Sistema de coordenadas universal del mundo.

El valor por defecto de este espacio de transformación depende de la operación a realizar. Por ejemplo, la rotación (ya sea mediante la llamada a *rotate* o a *pitch, yaw* o *roll*) se realiza por defecto con respecto al sistema de local, mientras que la traslación se realiza por defecto relativa al padre.

Veamos en el siguiente listado un ejemplo que ilustre estos conceptos. En el listado se definen tres nodos, en el que *node1* es el padre de *node2* y *node3*. A node 2 se le aplica una rotación respecto del eje y en la línea 15. Como hemos comentado anteriormente, por defecto (si no se le indica ningún parámetro adicional) se realiza sobre `TS_LOCAL`[1] Posteriormente se aplica una traslación en la línea 16, que por defecto se realiza relativa al sistema de coordenadas del nodo padre. Al *node3* se el aplican exactamente la misma rotación y traslación, pero ésta última se aplica respecto del sistema de referencia local (línea 23). Como puede verse en la Figura 10.7, la entidad adjunta al *node3* se representa trasladada 5 unidades respecto de su sistema de coordenadas local.

Listado 10.3: Uso de diferentes espacios de transformación

```
1 class SimpleExample : public ExampleApplication {
2  public : void createScene() {
```

[1]Especificar el valor por defecto del espacio de transformación tiene el mismo efecto que no especificarlo. Por ejemplo, cambiar la línea 15 del ejemplo por `node2->yaw(Degree(-90), Node::TS_LOCAL);` no tendría ningún efecto diferente sobre el resultado final.

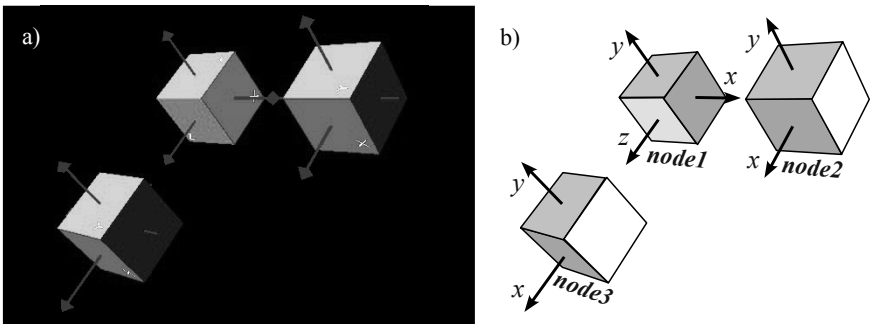


Figura 10.7: Ejemplo de trabajo con diferentes espacios de transformación. **a)** Resultado de ejecución del ejemplo. **b)** Distribución de los nodos del *Grafo de Escena*.

```
3    SceneNode* node1 = mSceneMgr->createSceneNode("Node1");
4    Entity *ent1 = mSceneMgr->createEntity("ent1", "cuboejes.mesh");
5    node1->attachObject(ent1);
6    mSceneMgr->getRootSceneNode()->addChild(node1);
7    node1->setPosition(0,0,480);
8    node1->yaw(Degree(-45));    // Por defecto es Node::TS_LOCAL
9    node1->pitch(Degree(45));   // Por defecto es Node::TS_LOCAL
10
11   SceneNode* node2 = mSceneMgr->createSceneNode("Node2");
12   Entity *ent2 = mSceneMgr->createEntity("ent2", "cuboejes.mesh");
13   node2->attachObject(ent2);
14   node1->addChild(node2);
15   node2->yaw(Degree(-90));   // Por defecto es Node::TS_LOCAL
16   node2->translate(5,0,0);   // Por defecto es Node::TS_PARENT
17
18   SceneNode* node3 = mSceneMgr->createSceneNode("Node3");
19   Entity *ent3 = mSceneMgr->createEntity("ent3", "cuboejes.mesh");
20   node3->attachObject(ent3);
21   node1->addChild(node3);
22   node3->yaw(Degree(-90));   // Por defecto es Node::TS_LOCAL
23   node3->translate(5,0,0, Node::TS_LOCAL);  // Cambiamos a LOCAL!
24  }
25 };
```

El orden de las operaciones resulta especialmente relevante cuando se trabaja con diferentes espacios de transformación. ¿Qué ocurriría por ejemplo si invertimos el orde de las líneas 21 y 22 del código anterior?. En ese caso, las entidades relativas al *node2* y al *node3* aparecerán desplegadas exactamente en el mismo lugar, como se muestra en la Figura 10.8.

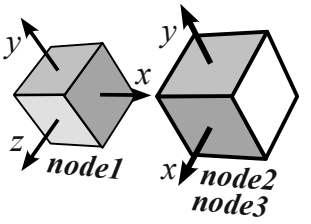


Figura 10.8: Tras aplicar la traslación de la línea 23, se aplica la rotación local al *node3*. Ahora el objeto del *node2* y *node3* aparecen alineados exactamente en la misma posición del espacio.

Capítulo 11

Recursos Gráficos y Sistema de Archivos

Javier Alonso Albusac Jiménez
Carlos González Morcillo

En este capítulo se realizará un análisis de los recursos que son necesarios en los gráficos 3D, centrándose sobre todo en los formatos de especificación y requisitos de almacenamiento. Además, se analizarán diversos casos de estudio de formatos populares como MD2, MD3, MD5, Collada y se hará especial hincapié en el formato de Ogre 3D.

11.1. Formatos de Especificación

11.1.1. Introducción

En la actualidad, el desarrollo de videojuegos de última generación implica la manipulación simultánea de múltiples ficheros de diversa naturaleza, como son imágenes estáticas (BMP (BitMaP), JPG, TGA (Truevision Graphics Adapter), PNG, ...), ficheros de sonido (WAV (WAVeform), OGG, MP3 (MPEG-2 Audio Layer III)), mallas que representan la geometría de los objetos virtuales, o secuencias de vídeo (AVI (Audio Video Interleave), BIK (BINK Video), etc). Una cuestión relevante es cómo se cargan estos ficheros y qué recursos son necesarios para su reproducción, sobre todo teniendo en cuenta que la reproducción debe realizarse en tiempo real, sin producir ningún tipo de demora que acabaría con la paciencia del usuario.

A la hora de diseñar un videojuego es muy importante tener en mente que los recursos hardware no son ilimitados. Las plataformas empleadas para su ejecución poseen diferentes prestaciones de memoria, potencia de procesamiento, tarjeta gráfica, etc. No todos los usuarios que adquieren un juego están dispuestos a mejorar las prestaciones de su equipo para poder ejecutarlo con ciertas garantías de calidad [62]. Por todo ello, es fundamental elegir los formatos adecuados para cada tipo de archivo, de tal forma que se optimice en la mayor medida de lo posible los recursos disponibles.

El formato empleado para cada tipo de contenido determinará el tamaño y este factor afecta directamente a uno de los recursos más importantes en la ejecución de un videojuego: la memoria [62]. Independientemente del tamaño que pueda tener la memoria de la plataforma donde se ejecuta el videojuego, ésta suele estar completa durante la ejecución. Durante la ejecución se producen diversos cambios de contexto, en los que se elimina el contenido actual de la memoria para incluir nuevos datos. Por ejemplo, cuando se maneja un personaje en una determinada escena y éste entra en una nueva habitación o nivel, los datos de las escena anterior son eliminados temporalmente de la memoria para cargar los nuevos datos correspondientes al nuevo lugar en el que se encuentra el personaje protagonista. Es entonces cuando entran en juego los mecanismos que realizan el intercambio de datos y gestionan la memoria. Lógicamente, cuanto menor sea el espacio que ocupan estos datos más sencillo y óptimo será el intercambio en memoria. A la hora de elegir formato para cada tipo de fichero siempre hay que tratar de encontrar un equilibrio entre calidad y tamaño.

En la mayoría de videojuegos, los bits seleccionados se empaquetan y comprimen para ser ejecutados en el momento actual; a estos archivos se les conoce como ficheros de recursos, los cuales contienen una amplia variedad de datos multimedia (imágenes, sonidos, mallas, mapas de nivel, vídeos, etc).

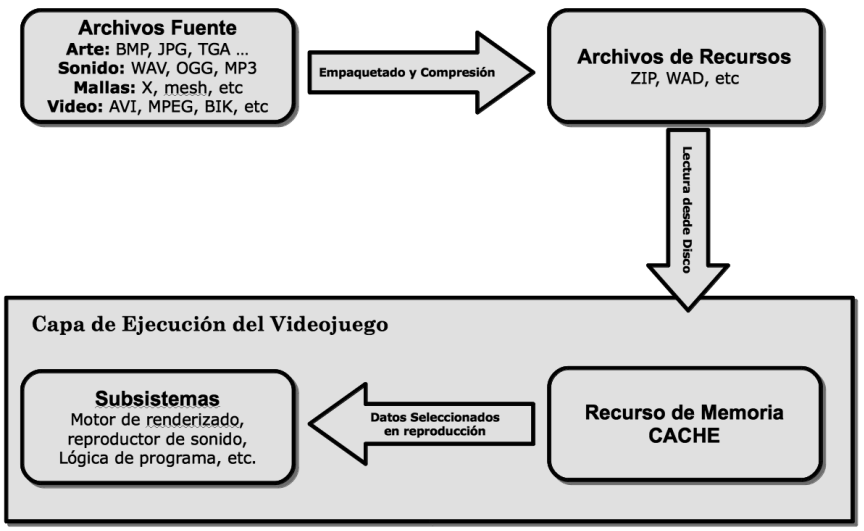


Figura 11.1: Flujo de datos desde los archivos de recursos hasta los subsistemas que se encargan de la reproducción de los contenidos [62].

Normalmente estos archivos están asociados a un nivel del juego. En cada uno de estos niveles se definen una serie de entornos, objetos, personajes, objetivos, eventos, etc. Al cambiar de nivel, suele aparecer en la pantalla un mensaje de carga y el usuario debe esperar; el sistema lo que está haciendo en realidad es cargar el contenido de estos ficheros que empaquetan diversos recursos.

Cada uno de los archivos empaquetados se debe convertir en un formato adecuado que ocupe el menor número de recursos posible. Estas conversiones dependen en la mayoría de los casos de la plataforma *hardware* en la que se ejecuta el juego. Por ejemplo, las plataformas PS3 y Xbox360 presentan formatos distintos para el sonido y las texturas de los objetos 3D.

En la siguiente sección se verá con mayor grado de detalle los recursos gráficos 3D que son necesarios, centrándonos en los formatos y los requisitos de almacenamiento.

11.1.2. Recursos de gráficos 3D: formatos y requerimientos de almacenamiento

Los juegos actuales con al menos una complejidad media-alta suelen ocupar varios GigaBytes de memoria. La mayoría de estos juegos constan de un conjunto de ficheros cerrados con formato privado que encapsulan múltiples contenidos, los cuales están distribuidos en varios DVDs (4.7 GB por DVD) o en un simple Blue-Ray (25GB) como es en el caso de los juegos vendidos para la plataforma de Sony - PS3. Si tenemos algún juego instalado en un PC, tenemos acceso a los directorios de instalación y podemos hacernos una idea del gran tamaño que ocupan una vez que los juegos han sido instalados.

Lo que vamos a intentar en las siguientes secciones es hacernos una idea de cómo estos datos se almacenan, qué formatos utilizan y cómo se pueden comprimir los datos para obtener el producto final. Por tanto, lo primero que debemos distinguir son los tipos de ficheros de datos que normalmente se emplean. La siguiente clasificación, ofrece una visión general [62]:

- **Objetos y mallas 3D para el modelado de entornos virtuales:** para el almacenamiento de este tipo de datos, normalmente son necesarias unas pocas decenas de megabytes para llevar a cabo dicha tarea. En este tipo de ficheros se almacena toda la geometría asociada al videojuego.
- **Mallas 3D y datos de animación:** estos datos en realidad no ocupan demasiado espacio pero suele suponer un número elevado de ficheros, la suma de todos ellos puede ocupar varias decenas de MB también.
- **Mapa/ Datos de nivel:** en este tipo de archivos se almacenan disparadores de eventos (eventos que se pueden producir en un determinado escenario y asociación de acciones a realizar una vez que se producen), Tipos de objetos del entorno, *scripts*, etc. No ocupan demasiado espacio al igual que el caso anterior, y suele ser bastante sencillo compactarlos o comprimirlos.
- **Sprites (personajes) y texturas asociadas a los materiales:** suele haber bastante información asociada a estos tipos de datos. En un juego medianamente complejo, los ficheros de este tipo comienzan enseguida a ocupar bastante espacio, hablamos de cientos de megas.
- **Sonido, música y diálogos:** suelen ser los datos que ocupan mas espacio de todo el juego, sobre todo cuando los juegos relatan una profunda y larga historia.
- **Vídeo y escenas pre-grabadas:** Cuando se usa este tipo de recursos suele ocupar la mayoría de espacio, por eso se usan con moderación. Suelen ser la combinación de personajes animados con archivos de sonido.

En las siguientes secciones se describirá con mayor detalle cada uno de los puntos anteriores.

Objetos y Mallas 3D

Al contrario de lo que normalmente se puede pensar, la geometría de los elementos tridimensionales que se emplean en un videojuego no ocupan demasiado espacio en comparación con otro tipo de contenidos [62]. Como se comentó en las secciones anteriores, los principales consumidores de espacio son los ficheros de audio y vídeo.

Una malla 3D, independientemente de que corresponda con un personaje, objeto o entorno, es una colección de puntos situados en el espacio, con una serie de datos asociados que describen cómo estos puntos están organizados y forman un conjunto de polígonos y cómo éstos deben ser renderizados.

Por otro lado, a los puntos situados en el espacio se les llama vértices y se representan mediante tres puntos pertenecientes a las tres coordenadas espaciales (X,Y,Z), tomando como referencia el punto de origen situado en (0,0,0). Cualquier elemento virtual se modela mediante triángulos que forman la malla, y cada triángulo se define por medio de tres o mas índices en una lista de puntos. Aquí se puede mostrar un ejemplo de una malla que representa un cubo.

Para representar los triángulos del cubo siguiente, necesitaríamos tres vértices por triángulo y tres coordenadas por cada uno de esos vértices. Existen diferentes formas de ahorrar espacio y optimizar la representación de triángulos. Si se tiene en cuenta que varios triángulos tienen vértices en común, no es necesario representar esos vértices más de una vez. El método consiste en representar únicamente los tres índices del primer triángulo y, para el resto de triángulos, únicamente se añade el vértice adicional. Esta técnica sería similar a dibujar el cubo con un lápiz sin separar en ningún momento la punta del lápiz del papel.

Listado 11.1: Representación de los vértices de un cubo con respecto al origen de coordenadas

```
Vec3 TestObject::g_SquashedCubeVerts[] =
{
    Vec3( 0.5,0.5,-0.25),      // Vertex 0.
    Vec3(-0.5,0.5,-0.25),      // Vertex 1.
    Vec3(-0.5,0.5,0.5),        // And so on.
    Vec3(0.75,0.5,0.5),
    Vec3(0.75,-0.5,-0.5),
    Vec3(-0.5,-0.5,-0.5),
    Vec3(-0.5,-0.3,0.5),
    Vec3(0.5,-0.3,0.5)
};
```

Listado 11.2: Representación de los triángulos de un cubo mediante asociación de índices

```
WORD TestObject::g_TestObjectIndices[][3] =
{
    { 0,1,2 },   { 0,2,3 },   { 0,4,5 },
    { 0,5,1 },   { 1,5,6 },   { 1,6,2 },
    { 2,6,7 },   { 2,7,3 },   { 3,7,4 },
    { 3,4,0 },   { 4,7,6 },   { 4,6,5 }
};
```

Con un simple cubo es complicado apreciar los beneficios que se pueden obtener con esta técnica, pero si nos paramos a pensar que un modelo 3D puede tener miles y miles de triángulos, la optimización es clara. De esta forma es posible almacenar *n* triángulos con *n+2* índices, en lugar de *n*3* vértices como sucede en el primer caso.

Además de la información relativa a la geometría del objeto, la mayoría de formatos soporta la asociación de información adicional sobre el material y textura de cada uno de los polígonos que forman el objeto. El motor de *rendering* asumirá, a menos que se indique lo contrario, que cada grupo de triángulos tiene asociado el mismo material y las mismas texturas. El material define el color de un objeto y como se refleja la luz en él. El tamaño destinado al almacenamiento de la información relativa al material puede variar dependiendo del motor de *rendering* empleado. Si al objeto no le afecta la luz directamente y tiene un color sólido, tan sólo serán necesarios unos pocos de bytes extra. Pero, por el contrario, si al objeto le afecta directamente la luz y éste tiene una textura asociada, podría suponer casi 100 bytes más por cada vértice.

De todo esto debemos aprender que la geometría de un objeto puede ocupar mucho menos espacio si se elige un formato adecuado para representar los triángulos que forman el objeto. De cualquier forma, los requisitos de memoria se incrementan notablemente cuando se emplean materiales complejos y se asocian texturas al objeto.

Datos de Animación

Una animación es en realidad la variación de la posición y la orientación de los vértices que forman un objeto a lo largo del tiempo. Como se comentó anteriormente, una forma de representar una posición o vértice en el espacio es mediante tres valores reales asociados a las tres coordenadas espaciales X, Y y Z. Estos números se representan siguiendo el estándar IEEE (Institute of Electrical and Electronics Engineers)-754 para la representación de números reales en coma flotante mediante 32 bits (4 bytes). Por tanto, para representar una posición 3D será necesario emplear 12 bytes.

Además de la posición es necesario guardar información relativa a la orientación, y el tamaño de las estructuras de datos empleadas para ello suele variar entre 12 y 16 bytes, dependiendo del motor de *rendering*. Existen diferentes formas de representar la orientación, el método elegido influirá directamente en la cantidad de bytes necesarios. Dos de los métodos más comunes en el desarrollo de videojuegos son los ángulos de Euler y el uso de cuaterniones o también llamados cuaternios.

Para hacernos una idea del tamaño que sería necesario en una sencilla animación, supongamos que la frecuencia de reproducción de frames por segundo es de 25. Por otro lado, si tenemos en cuenta que son necesarios 12 bytes por vértice más otros 12 (como mínimo) para almacenar la orientación de cada vértice, necesitaríamos por cada vértice 12 + 12 = 24 bytes por cada frame. Si en un segundo se reproducen 25 frames, 25 x 24 = 600 bytes por cada vértice y cada segundo. Ahora supongamos que un objeto consta de 40 partes movibles (normalmente las partes movibles de un personaje las determina el esqueleto y los huesos que lo forman). Si cada parte movible necesita 600 bytes y existen 40 de ellas, se necesitaría por cada segundo un total de 24.000 bytes.

Naturalmente 24.000 bytes puede ser un tamaño excesivo para un solo segundo y existen diferentes formas de optimizar el almacenamiento de los datos de animación, sobre todo teniendo en cuenta que no todas las partes del objeto se mueven en cada momento y, por tanto, no sería necesario almacenarlas de nuevo. Tampoco es necesario almacenar la posición de cada parte movible en cada uno de los 25 frames que se reproducen en un segundo. Una solución elegante es establecer frames claves y calcular las posiciones intermedias de un vértice desde un frame clave al siguiente mediante interpolación lineal. Es decir, no es necesario almacenar todas las posiciones (únicamente la de los frames claves) ya que el resto pueden ser calculadas en tiempo de ejecución.

Otra posible mejora se podría obtener empleando un menor número de bytes para representar la posición de un vértice. En muchas ocasiones, el desplazamiento o el cambio de orientación no son exageradamente elevados y no es necesario emplear 12 bytes. Si los valores no son excesivamente elevados se pueden emplear, por ejemplo, números enteros de 2 bytes. Estas técnicas de compresión pueden reducir considerablemente el tamaño en memoria necesario para almacenar los datos de animación.

Mapas/Datos de Nivel

Cada uno de los niveles que forman parte de un juego tiene asociado diferentes elementos como objetos estáticos 3D, sonido de ambientación, diálogos, entornos, etc. Normalmente, todos estos contenidos son empaquetados en un fichero binario que suele ser de formato propietario. Una vez que el juego es instalado en nuestro equipo es complicado acceder a estos contenidos de manera individual. En cambio, durante el proceso de desarrollo estos datos se suelen almacenar en otros formatos que sí son accesibles por los miembros del equipo como, por ejemplo, en formato XML. El formato XML es un formato accesible, interpretable y permite a personas que trabajan con diferentes herramientas y en diferentes ámbitos, disponer de un medio para intercambiar información de forma sencilla.

Texturas

Hasta el momento, los datos descritos para el almacenamiento de la geometría de los objetos, animación y datos de nivel no suponen un porcentaje alto de la capacidad de almacenamiento requerida. Las texturas (en tercer lugar), los ficheros de audio y vídeo son los que implican un mayor coste en términos de memoria.

La textura es en realidad una imagen estática que se utiliza como "piel" para cubrir la superficie de un objeto virtual. Existen multitud de formatos que permiten obtener imágenes de mayor o menor calidad y, en función de esta calidad, de un mayor o menor tamaño. Para obtener la máxima calidad, los diseñadores gráficos prefieren formatos no comprimidos de 32 bits como TIF (TIFF) o TGA. Por contra, el tamaño de estas imágenes podría ser excesivo e influir en el tiempo de ejecución de un videojuego. Por ejemplo, una imagen RAW (imagen sin modificaciones, sin comprimir) de 32 bits con una resolución de 1024 x 768 píxeles podría alcanzar el tamaño de 3MB. Por tanto, una vez más es necesario encontrar un equilibrio entre calidad y tamaño. Cuando se diseña un videojuego, una de las principales dificultades que se plantean es la elección de un formato adecuado para cada uno de los recursos, tal que se satisfagan las expectativas de calidad y eficiencia en tiempo de ejecución.

Uno de los parámetros característicos de cualquier formato de imagen es la profundidad del color [62]. La profundidad del color determina la cantidad de bits empleados para representar el color de un píxel en una imagen digital. Si con n bits se pueden representar 2^n valores, el uso de n bits por píxel ofrecerá la posibilidad de representar 2^n colores distintos, es decir, cuando mayor sea el número n de bits empleados, mayor será la paleta de colores disponibles.

- **32-bits (8888 RGBA (Red Green Blue Alpha)).** Las imágenes se representan mediante cuatro canales, R(red), G(green), B(blue), A(alpha), y por cada uno de estos canales se emplean 8 bits. Es la forma menos compacta de representar un mapa de bits y la que proporciona un mayor abanico de colores. Las imágenes representadas de esta forma poseen una calidad alta, pero en muchas ocasiones es innecesaria y otros formatos podrían ser más apropiados considerando que hay que ahorrar el mayor número de recursos posibles.

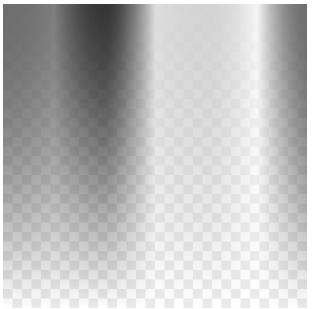


Figura 11.2: Ejemplo de una imagen RGBA con porciones transparentes (canal alpha).

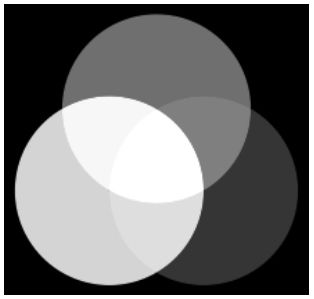


Figura 11.3: Modelo aditivo de colores rojo, verde y azul.

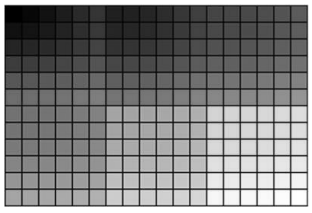


Figura 11.4: Paleta de 256 colores con 8 bits

- **24-bits (888 RGB (Red Green Blue))**. Este formato es similar al anterior pero sin canal alpha, de esta forma se ahorran 8 bits y los 24 restantes se dividen en partes iguales para los canales R(red), G(green) y B(blue). Este formato se suele emplear en imágenes de fondo que contienen una gran cantidad de colores que no podrían ser representados con 16 u 8 bits.
- **24-bits (565 RGB, 8A).** Este formato busca un equilibrio entre los dos anteriores. Permite almacenar imágenes con una profundidad de color aceptable y un rango amplio de colores y, además, proporciona un canal alfa que permite incluir porciones de imágenes traslúcidas. El canal para el color verde tiene un bit extra debido a que el ojo humano es más sensible a los cambios con este color.
- **16-bits (565 RGB).** Se trata de un formato compacto que permite almacenar imágenes con diferentes variedades de colores sin canal alfa. El canal para el color verde también emplea un bit extra al igual que en el formato anterior.
- **16-bits (555 RGB, 1 A)**. Similar al formato anterior, excepto el bit extra del canal verde que, ahora, es destinado al canal alfa.
- **8-bits indexado**. Este formato se suele emplear para representar iconos o imágenes que no necesitan alta calidad, debido a que la paleta o el rango de colores empleado no es demasiado amplio. Sin embargo, son imágenes muy compactas que ahorran mucho espacio en memoria y se transmiten o procesan rápidamente. En este caso, al emplearse 8 bits, la paleta sería de 256 colores y se representa de forma matricial, donde cada color tiene asociado un índice. Precisamente los índices son los elementos que permiten asociar el color de cada píxel en la imagen a los colores representados en la paleta de colores.

11.1.3. Casos de Estudio

Formato MD2/MD3

El formato MD2 es uno de los más populares por su simplicidad, sencillez de uso y versatilidad para la creación de modelos (personajes, entornos, armas, efectos, etc). Fue creado por la compañía id Software y empleado por primera vez en el videojuego Quake II (ver Figura 11.5).

El formato MD2 representa la geometría de los objetos, texturas e información sobre la animación de los personajes en un orden muy concreto, tal como muestra la Figura 11.6 [72]. El primer elemento en la mayoría de formatos 3D es la cabecera. La cabecera se sitúa al comienzo del fichero y es realmente útil ya que contiene información relevante sobre el propio fichero sin la necesidad de tener que buscarla a través de la gran cantidad de datos que vienen a continuación.

Cabecera

La cabecera del formato MD2 contiene los siguientes campos:

Listado 11.3: Cabecera del formato MD2

```
1 struct SMD2Header {
2     int m_iMagicNum;
3     int m_iVersion;
4     int m_iSkinWidthPx;
5     int m_iSkinHeightPx;
6     int m_iFrameSize;
7     int m_iNumSkins;
8     int m_iNumVertices;
9     int m_iNumTexCoords;
```

Figura 11.5: Captura del vídeojuego Quake II, desarrollado por la empresa id Software, donde se empleó por primera vez el formato MD2

```
10      int m_iNumTriangles;
11      int m_iNumGLCommands;
12      int m_iOffsetSkins;
13      int m_iOffsetTexCoords;
14      int m_iOffsetTriangles;
15      int m_iOffsetFrames;
16      int m_iOffsetGlCommands;
17      int m_iFileSize;
18      int m_iNumFrames;
19
20  };
```

El tamaño total es de 68 bytes, debido a que cada uno de los enteros se representa haciendo uso de 4 bytes. A continuación, en el siguiente listado se describirá brevemente el significado de cada uno de estos campos [72].

- En primer lugar podemos apreciar un campo que se refiere al "número mágico". Este número sirve para identificar el tipo de archivo y comprobar que en efecto se trata de un fichero con formato MD2.
- Versión del fichero. De esta forma se lleva a cabo un control de versiones y se evita el uso de versiones obsoletas del fichero.
- La siguiente variable hace referencia a la textura que se utiliza para cubrir la superficie del modelo. En este formato, cada modelo MD2 puede utilizar un piel o textura en un instante concreto de tiempo, aunque se podrían cargar varias texturas e ir alternándolas a lo largo del tiempo.
- *m_iSkinWidthPx* y *m_iSkinHeightPx* representan la anchura y altura de la textura en píxeles.

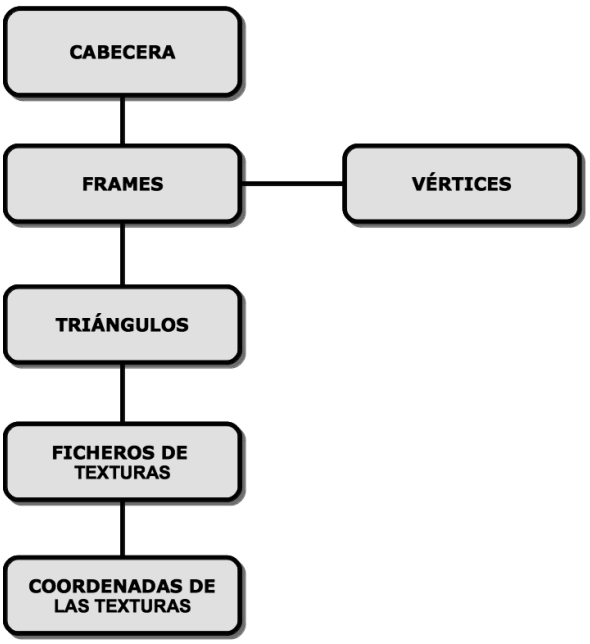


Figura 11.6: Jerarquía de capas en un fichero con formato MD2

- *m_iFrameSize* es un entero que representa el tamaño en bytes de cada frame clave.

Las seis variables siguientes se refieren a cantidades [72]:

- *m_iNumSkins* es el número de texturas definidas en el fichero. Como se comentó anteriormente, el formato no soporta la aplicación simultánea de múltiples texturas, pero sí cargar varias texturas y ser aplicadas en diferentes instantes.
- *m_iNumVertices* es un entero que representa el número de vértices por frame.
- *m_iNumTexCoords* representa el número de coordenadas de la textura. El número no tiene que coincidir con el número de vértices y se emplea el mismo número de coordenadas para todos los frames.
- *m_iNumTriangles* es el número de triángulos que compone el modelo. En el formato MD2, cualquier objeto está formado mediante la composición de triángulos, no existe otro tipo de primitiva como, por ejemplo, cuadriláteros.
- *m_iNumGLCommands* especifica el número de comandos especiales para optimizar el renderizado de la malla del objeto. Los comandos GL no tienen por qué cargar el modelo, pero sí proporcionan una forma alternativa de renderizarlo.
- *m_iNumFrames* representa el número de frames en el fichero MD2 file. Cada uno de los frames posee información completa sobre las posiciones de los vértices para el proceso de animación.

Las cinco variables siguientes se emplean para definir el *offset* o dirección relativa (el número de posiciones de memoria que se suman a una dirección base para obtener la dirección absoluta) en bytes de cada una de las partes del fichero. Esta información es fundamental para facilitar los saltos en las distintas partes o secciones del fichero durante el proceso de carga. Por último, la variable *m_iFileSize* representa el tamaño en bytes desde el inicio de la cabecera hasta el final del fichero.

Frames y Vértices

A continuación se muestra la estructura que representa la información que se maneja por frame en el formato MD2 [72]:

Listado 11.4: Información empleada en cada frame en el formato MD2

```
1  struct SMD2Frame {
2      float m_fScale[3];
3      float m_fTrans[3];
4      char m_caName[16];
5      SMD2Vert * m_pVertss
6
7      SMD2Frame()
8      {
9          m_pVerts = 0;
10     }
11     ~SMD2Frame()
12     {
13         if(m_pVerts) delete [] m_pVerts;
14     }
15 };
```

Como se puede apreciar en la estructura anterior, cada frame comienza con seis números representados en punto flotante que representan la escala y traslación de los vértices en los ejes X, Y y Z. Posteriormente, se define una cadena de 16 caracteres que determinan el nombre del frame, que puede resultar de gran utilidad si se quiere hacer referencia a éste en algún momento. Por último se indica la posición de todos los vértices en el frame (necesario para animar el modelo 3D). En cada uno de los frames habrá tantos vértices como se indique en la variable de la cabecera *m_iNumVerts*. En la última parte de la estructura se puede diferenciar un constructor y destructor, que son necesarios para reservar y liberar memoria una vez que se ha reproducido el frame, ya que el número máximo de vértices que permite almacenar el formato MD2 por frame es 2048.

Si se renderizaran únicamente los vértices de un objeto, en pantalla tan sólo se verían un conjunto de puntos distribuidos en el espacio. El siguiente paso consistirá en unir estos puntos para definir los triángulos y dar forma al modelo.

Triangularización

En el formato MD2 los triángulos se representan siguiendo la técnica descrita en la Sección 11.1.2. Cada triángulo tiene tres vértices y varios triángulos tienen índices en común [72]. No es necesario representar un vértice más de una vez si se representa la relación que existe entre ellos mediante el uso de índices en una matriz.

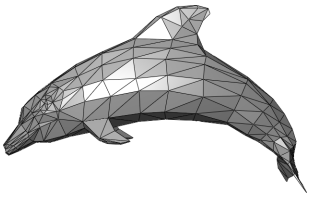


Figura 11.7: Modelado de un objeto tridimensional mediante el uso de triángulos.

Además, cada triángulo debe tener asociado una textura, o sería más correcto decir, una parte o fragmento de una imagen que representa una textura. Por tanto, es necesario indicar de algún modo las coordenadas de la textura que corresponden con cada triángulo. Para asociar las coordenadas de una textura a cada triángulo se emplean el mismo número de índices, es decir, cada índice de un vértice en el triángulo tiene asociado un índice en el array de coordenadas de una textura. De esta forma se puede texturizar cualquier objeto fácilmente.

A continuación se muestra la estructura de datos que se podía emplear para representar los triángulos mediante asociación de índices en un array y la asociación de coordenadas de la textura para cada vértice:

Listado 11.5: Estructura de datos para representar los triángulos y las coordenadas de la textura

```
1 struct SMD2Tri {
2     unsigned short m_sVertIndices[3];
3     unsigned short m_sTexIndices[3];
4 };
```

Inclusión de Texturas

En el formato MD2 existen dos formas de asociar texturas a los objetos. La primera de ellas consiste en incluir el nombre de las texturas embebidos en el fichero MD2. El número de ficheros de texturas y la localización, se encuentran en las variables de la cabecera *m_iNumSkins* y *m_iOffsetSkins*. Cada nombre de textura ocupa 64 caracteres alfanuméricos como máximo. A continuación se muestra la estructura de datos empleada para definir una textura [72]:

Listado 11.6: Estructura de datos para definir una textura embebida en un fichero MD2

```
1 struct SMD2Skin
2 {
3     char m_caSkin[64];
4     CImage m_Image;
5 };
```

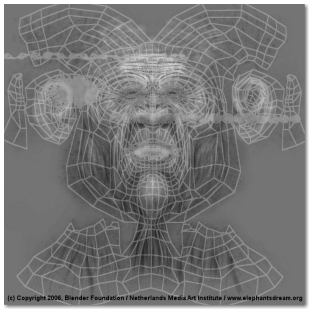


Figura 11.8: Ejemplo de textura en la que se establece una correspondencia entre los vértices del objeto y coordenadas de la imagen.

Tal como se puede apreciar en la estructura anterior, existe una instancia de la clase *CImage*. Esta clase posee varias funciones para cargar y asociar varias clases de texturas. Existe una textura diferente por cada nombre de fichero que aparece en la sección de texturas dentro del fichero MD2.

Una segunda alternativa es utilizar la clase *CImage* y el método *SetSkin* para cargar una textura y asociarla a un objeto [72]. Si existe una correspondencia por coordenadas entre la textura y el objeto, será necesario cargar las coordenadas antes de asociar la textura al objeto. El número de coordenadas de la textura se puede encontrar en la variable de la cabecera *m_iNumTexCoords*. La estructura de datos que se utiliza para definir las coordenadas de una textura es la siguiente:

Listado 11.7: Estructura de datos para representar las coordenadas de una textura

```
1 struct SMD2TexCoord {
2     float m_fTex[2];
3 };
```

Cada coordenada de una textura consiste en un par de números de tipo *float* de 2 bytes. La primera coordenada de la textura comienza en el cero y finaliza en el valor equivalente a la anchura de la imagen; la segunda desde el cero hasta el valor de la altura. En el resto de coordenadas se establecen las correspondencias entre zonas de la imagen de la textura y las superficies del objeto. Finalmente, una vez que se han definido las coordenadas ya se puede asociar la textura al objeto, mediante la función *Bind* de la clase *CImage*.

Principales diferencias entre el formato MD2 y MD3

El formato MD3 fue el formato que se empleó en el desarrollo del videojuego Quake III y sus derivados (Q3 mods, Return to Castle Wolfenstein, Jedi Knights 2, etc.). MD3 se basa en su predecesor pero aporta mejoras notables en dos aspectos claves:

- **Animación de personajes.** La frecuencia de reproducción de frames en el formato MD2 está limitada a 10 fps. En el caso del formato MD3 la frecuencia es variable, permitiendo animaciones de vértices más complejas.
- **Modelado de objetos 3D**. Otra diferencia significativa entre los formatos MD2 y MD3 es que en este último los objetos se dividen en tres bloques diferenciados, normalmente, cabeza, torso y piernas. Cada uno de estos bloques se tratan de manera independiente y esto implica que cada parte tenga su propio conjunto de texturas y sean renderizados y animados por separado.

Figura 11.9: Captura del vídeojuego Quake III.

Formato MD5

El formato MD5 se empleó en el desarrollo de los videojuegos Doom III y Quake IV. El formato presenta mejoras significativas en la animación de personajes. En el caso de los formatos MD2 y MD3 los movimientos de cada personaje se realizaban mediante animación de vértices, es decir, se almacenaba la posición de los vértices de un personaje animado en cada frame clave. Tratar de forma individualizada cada vértice es realmente complejo y animar un personaje siguiendo esta metodología no es una tarea sencilla.

En el formato MD5 es posible definir un esqueleto formado por un conjunto de huesos y asociarlo a un personaje. Cada uno de los huesos tiene a su vez asociado un conjunto de vértices. La animación en el formato MD5 se basa en la animación de los huesos que forman el esqueleto; el grupo de vértices asociado a un hueso variará su posición en base al movimiento de éste. Una de las grandes ventajas de la animación mediante movimiento de huesos de un esqueleto es que ésta se puede almacenar y reutilizar para personajes que tengan un esqueleto similar.

COLLADA

El formato COLLADA (COLLAborative Design Activity) [43] surge ante la necesidad de proponer un formato estándar de código abierto que sirva como medio de intercambio en la distribución de contenidos. La mayoría de empresas utilizan su propio formato de código cerrado y en forma binaria, lo que dificulta el acceso a los datos y la reutilización de contenidos por medio de otras herramientas que permiten su edición. Con la elaboración de COLLADA se pretenden alcanzar una serie de objetivos básicos [68]:

- COLLADA no es un formato para motores de videojuegos. En realidad, COLLADA beneficia directamente a los usuarios de herramientas de creación y distribución de contenidos. Es decir, COLLADA es un formato que se utiliza en el proceso de producción como mecanismo de intercambio de información entre los miembros del equipo de desarrollo y no como mecanismo final de producción.
- El formato COLLADA debe ser independiente de cualquier plataforma o tecnología de desarrollo (sistemas operativos, lenguajes de programación, etc).

- Ser un formato de código libre, representado en XML para liberar los recursos digitales de formatos binarios propietarios.
- Proporcionar un formato estándar que pueda ser utilizado por una amplia mayoría de herramientas de edición de modelos tridimensionales.
- Intentar que sea adoptado por una amplia comunidad de usuarios de contenidos digitales.
- Proveer un mecanismo sencillo de integración, tal que toda la información posible se encuentre disponible en este formato.
- Ser la base común de todas las transferencias de datos entre aplicaciones 3D.

Cualquier fichero XML representado en el formato COLLADA está dividido en tres partes principales [68]:

- **COLLADA Core Elements**. Donde se define la geometría de los objetos, información sobre la animación, cámaras, luces, etc.
- **COLLADA Physics**. En este apartado es posible asociar propiedades físicas a los objetos, con el objetivo de reproducir comportamientos lo más realistas posibles.
- **COLLADA FX.** Se establecen las propiedades de los materiales asociados a los objetos e información valiosa para el renderizado de la escena.

No es nuestro objetivo ver de forma detallada cada uno de estos tres bloques, pero sí se realizará una breve descripción de los principales elementos del núcleo ya que éstos componen los elementos básicos de una escena, los cuales son comunes en la mayoría de formatos.

COLLADA Core Elements

En este bloque se define la escena (<*scene*>) donde transcurre una historia y los objetos que participan en ella, mediante la definición de la geometría y la animación de los mismos. Además, se incluye información sobre las cámaras empleadas en la escena y la iluminación. Un documento con formato COLLADA sólo puede contener un nodo <*scene*>, por tanto, será necesario elaborar varios documentos COLLADA para definir escenas diferentes. La información de la escena se representa mediante un grafo acíclico y debe estar estructurado de la mejor manera posible para que el procesamiento sea óptimo. A continuación se muestran los nodos/etiquetas relacionados con la definición de escenas [68].

Listado 11.8: Nodos empleados en COLLADA para la definición de escenas

```
1 <instance_node>
2 <instance_visual_scene>
3 <library_nodes>
4 <library_visual_scenes>
5 <node>
6 <scene>
7 <visual_scene>
```

Para facilitar el manejo de una escena, COLLADA ofrece la posibilidad de agrupar los elementos en librerías. Un ejemplo sencillo de definición de una escena podría ser el siguiente:

Al igual que en el caso anterior, la definición de la geometría de un objeto también puede ser estructurada y dividida en librerías, muy útil sobre todo cuando la geometría de un objeto posee una complejidad considerable. De esta forma los nodos *geometry* se pueden encapsular dentro de una librería de objetos denominada *library_geometrics*.

Listado 11.9: Ejemplo de definición de una escena en el formato COLLADA

```
<COLLADA>
<library_nodes id="comp">
  <node name="earth">
  </node>
  <node name="sky">
  </node>
</library_nodes>

<library_visual_scenes>
  <visual_scene id="world">
   <instance_library_nodes url="#comp">
  </visual_scene>
</library_visual_scenes>

<scene>
  <instance_visual_scene url="#world"/>
</scene>
</COLLADA>
```

El elemento <*geometry*> permite definir la geometría de un objeto. En el caso de COLLADA (y en la mayoría de formatos) la geometría se define mediante la definición de una malla que incluye información del tipo: cantidad de puntos, posición, información de color, coordenadas de la textura aplicada al objeto, líneas que los conectan, ángulos, superficies, etc. Los nodos que se emplean para definir la geometría de un objeto en COLLADA son los siguientes [68]:

Listado 11.10: Nodos utilizados para definir la geometría de un objeto en el formato COLLADA

```
<control_vertices>
<geometry>
<instance_geometry>
<library_geometries>
<lines>
<linestrips>
<mesh>
<polygons>
<polylist>
<spline>
<triangles>
<trifans>
```

Un ejemplo de definición de una malla en el formato COLLADA podría ser el que se muestra en el siguiente listado [68].

Para completar la información de la geometría de un objeto es necesario incluir datos sobre la transformación de los vértices. Algunas de las operaciones más comunes son la rotación (<*rotate*>), escalado (<*scale*>) o traslaciones (<*translate*>).

Listado 11.11: Ejemplo de definición de una malla en el formato COLLADA

```
<mesh>
```

```
  <source id="position" />
  <source id="normal" />
  <vertices id="verts">
    <input semantic="POSITION" source="#position"/>
  </vertices>
  <polygons count="1" material="Bricks">
    <input semantic="VERTEX" source="#verts" offset="0"/>
    <input semantic="NORMAL" source="#normal" offset="1"/>
    <p>0 0  2 1  3 2  1 3</p>
  </polygons>
</mesh>
```

En cuanto a la iluminación, COLLADA soporta la definición de las siguientes fuentes de luces [68]:

- Luces ambientales
- Luces puntuales
- Luces direccionales
- Puntos de luz

y se utilizan los siguientes nodos o etiquetas:

Listado 11.12: Nodos empleados en COLLADA para la definición de fuentes de luz

```
<ambient>
<color>
<directional>
<instance_light>
<library_lights>
<Light>
<point>
<spot>
```

Por otro lado, COLLADA también permite la definición de cámaras. Una cámara declara una vista de la jerarquía del grafo de la escena y contiene información sobre la óptica (perspectiva u ortográfica). Los nodos que se utilizan para definir una cámara son los siguientes [68]:

Listado 11.13: Nodos empleados en COLLADA para definir cámaras

```
<camera>
<imager>
<instance_camera>
<library_cameras>
<optics>
<orthographic>
<Perspective>
```

Un ejemplo de definición de una cámara podría ser el siguiente [68]:

Listado 11.14: Ejemplo de definición de una cámara en el formato COLLADA

```
<camera name="eyepoint">
  <optics>
    <technique_common>
      <perspective>
        <yfov>45</yfov>
           <aspect_ratio>1.33333
```

```
7             </aspect_ratio>
8           <znear>1.0</znear>
9           <zfar>1000.0</zfar>
10        </perspective>
11      </technique_common>
12    </optics>
13  </camera>
```

En cuanto a la parte de animación no entraremos en detalle en esta sección ya que se verá en capítulos posteriores. Simplemente recalcar que COLLADA está preparado para soportar los dos tipos de animación principales: animación de vértices y animación de esqueletos asociado a objetos.

Formato para OGRE 3D

Los tres elementos esenciales en el formato de Ogre son los siguientes [45]:

- **Entity o entidad.** Una entidad es cualquier elemento que se puede dibujar en pantalla; por tanto, quedan excluidos de esta categoría las fuentes de luz y las cámaras. La posición y la orientación de una malla no se controla mediante este tipo de objeto pero sí el material y la textura asociada.
- **SceneNode o nodo de la escena**. Un nodo se utiliza para manipular las propiedades o principales características de una entidad (también sirven para controlar las propiedades de las fuentes de luz y cámaras). Los nodos de una escena están organizados de forma jerárquica y la posición de cada uno de ellos siempre es relativa a la de los nodos padre.
- **SceneManager o controlador de la escena**. Es el nodo raíz y de él derivan el resto de nodos que se dibujan en la escena. A través de este nodo es posible acceder a cualquier otro nodo en la jerarquía.

Si analizamos cualquier fichero OgreXML podemos apreciar que existe información relativa a las mallas de los objetos. Cada malla puede estar formada a su vez por una o varias submallas, que contienen la siguiente información:

- Definición de caras (<face>).
- Definición de vértices (<vertex>) con su posición, normal, color y coordenadas UV.
- Asignación de vértices a huesos (<vertexboneassignment>).
- Enlace a un esqueleto (<skeletonlink>).

A su vez cada submalla contiene el nombre del material y el número de caras asociada a la misma. Para cada una de las caras se almacenan los vértices que la componen:

Listado 11.15: Creación de mallas y submallas con materiales asociados

```
1 <mesh>
2     <submeshes>
3      <submesh material="blanco_ojo" usesharedvertices="false">
4          <faces count="700">
5              <face v1="0" v2="1" v3="2"/>
6             ..........................
7         </faces>
```

Además es necesario saber la información geométrica de cada uno de los vértices, es decir, posición, normal, coordenada de textura y el número de vértices de la submalla

Listado 11.16: Geometría de un Objeto en OgreXML

```
<geometry vertexcount="3361">
 <vertexbuffer positions="true" normals="true" texture_coords="1">
  <vertex>
   <position x="-0.000000" y="0.170180" z="-0.000000"/>
   <normal x="0.000000" y="1.000000" z="0.000000"/>
   <texcoord u="0.000000" v="1.000000"/>
  </vertex>
```

Para la asignación de los huesos se indican los huesos y sus pesos. Como se puede observar, el índice utilizado para los huesos empieza por el 0.

Listado 11.17: Asignación de huesos a una malla en OgreXML

```
<boneassignments>
 <vertexboneassignment vertexindex="0" boneindex="26" weight="
     1.000000"/>
 ..........................
</boneassignments>
</submesh>
```

Si la malla o mallas que hemos exportado tienen asociado un esqueleto se mostrará con la etiqueta "skeletonlink". El campo *name* corresponde con el archivo xml que contiene información sobre el esqueleto: *<skeletonlink name=çuerpo.skeleton/>*. En este caso el archivo "name.skeleton.xml"define el esqueleto exportado, es decir, el conjunto de huesos que componen el esqueleto. El exportador asigna a cada hueso un índice empezando por el 0 junto con el nombre, la posición y rotación del mismo:

Listado 11.18: Definición de un esqueleto en OgreXML

```
<skeleton>
<bones>
 <bone id="0" name="cerrada">
  <position x="5.395440" y="6.817142" z="-0.132860"/>
  <rotation angle="0.000000">
   <axis x="1.000000" y="0.000000" z="0.000000"/>
  </rotation>
 </bone>
 ..........................
</bones>
```

La definición del esqueleto no estaría completa si no se conoce la jerarquía de los huesos, esta información se describe indicando cuál es el padre de cada uno e los huesos:

Listado 11.19: Definición de la jerarquía de huesos en un esqueleto en el formato OgreXML

```
<bonehierarchy>
     <boneparent bone="torso" parent="caderas" />
     <boneparent bone="rota_ceja_izquierda" parent="ceja_izquierda"
         />
     <boneparent bone="rota_ceja_derecha" parent="ceja_derecha" />
     <boneparent bone="pecho" parent="torso" />
     <boneparent bone="hombro.r" parent="pecho" />
    ..........................
</bonehierarchy>
```

Por último, si se han creado animaciones asociadas al esqueleto y han sido exportadas, se mostrará cada una de ellas identificándolas con el nombre definido previamente en Blender y la longitud de la acción medida en segundos. Cada animación contendrá información sobre los huesos que intervienen en el movimiento (traslación, rotación y escalado) y el instante de tiempo en el que se ha definido el *frame* clave:

Listado 11.20: Animación de los huesos de un esqueleto en OgreXML

```
<animations>
 <animation name="ascensor" length="2.160000">
  <track bone="d1root.l">
   <keyframes>
   <keyframe time="0.000000">
    <translate x="-0.000000" y="0.000000" z="0.000000"/>
    <rotate angle="0.000000">
     <axis x="-0.412549" y="0.655310" z="0.632749"/>
    </rotate>
    <scale x="1.000000" y="1.000000" z="1.000000"/>
   </keyframe>

    ..........................
   <keyframe time="2.160000">
    <translate x="0.000000" y="-0.000000" z="0.000000"/>
      <rotate angle="0.000000">
       <axis x="-0.891108" y="0.199133" z="0.407765"/>
      </rotate>
      <scale x="1.000000" y="1.000000" z="1.000000"/>
   </keyframe>
  </keyframes>
  ..........................
</track>
```

Otros Formatos

A continuación se incluye una tabla con alguno de los formatos más comunes para la representación de objetos 3D, su extensión, editores que lo utilizan y un enlace donde se puede obtener más información sobre el formato (ver Tabla 3.1) [72].

11.2. Exportación y Adaptación de Contenidos

Como se comentó en secciones anteriores la mayoría de herramientas de creación de contenidos digitales posee su propio formato privado. Sin embargo, la mayoría de estas herramientas permiten la exportación a otros formatos para facilitar la distribución de contenidos. En esta sección nos centraremos en la creación de un modelo en Blender, la aplicación de texturas a dicho modelo mediante la técnica de UV Mapping y su exportación a los formatos XML y binario de Ogre. Finalmente, cargaremos el objeto exportado en una aplicación de Ogre 3D.

11.2.1. Instalación del exportador de Ogre en Blender

En primer lugar es necesario descargar la última versión del exportador desde *http://code.google.com/p/blender2ogre/*. En la sección *downloads* están disponibles las versiones del exportador asociadas a cada versión de blender. En función de la versión actual de blender que tengamos instalada, elegimos una u otra.

EXT.	EDITOR	Link
3DMF	3D Meta File, QuickDraw3D	http://www.apple.com
3DO	Jedi Knight	http://www.lucasarts.com
3DS	3D Studio Max, formato binario	http://www.discreet.com
ACT	Motor Genesis 3D	http://www.genesis3D.com
ASE	3D Studio Max: versión de 3DS basada en texto	http://www.discreet.com
ASC	3D Studio Max: mínima representación de datos representada en ASCII	http://www.discreet.com
B3D	Bryce 3D	http://www.corel.com
BDF	Okino	http://www.okino.com
BLEND	Blender	http://www.blender.com
CAR	Carrara	http://www.eovia.com/carrara
COB	Calgari TrueSpace	http://www.calgari.com
DMO	Duke Nukem 3D	http://www.3drealms.com
DXF	Autodesk Autocad	http://www.autodesk.com
HRC	Softimage 3D	http://www.softimage.com
INC	POV-RAY	http://www.povray.org
KF2	Animaciones y poses en Max Payne	http://www.maxpayne.com
KFS	Max Payne: información de la malla y los materiales	http://www.maxpayne.com
LWO	Lightwave	http://www.newtek.com
MB	Maya	http://www.aliaswavefront.com
MAX	3D Studio Max	http://www.discreet.com
MS3D	Milkshape 3D	http://www.swissquake.ch
OBJ	Alias|Wavefront	http://aliaswavefront.com
PZ3	Poser	http://www.curioslab.com
RAW	Triángulos RAW	http://
RDS	Ray Dream Studio	http://www.metacreations.com
RIB	Renderman File	http://www.renderman.com
VRLM	Lenguaje para el modelado de realidad virtual	http://www.web3d.org
X	Formato Microsoft Direct X	http://www.microsoft.com
XGL	Formato que utilizan varios programas CAD	http://www.xglspec.com

Cuadro 11.1: Otros formatos para la representación de objetos en 3D

Una vez descargado el archivo, lo descomprimimos y ejecutamos Blender. Posteriormente nos dirigimos al menú *file >user preferences* o bien pulsamos Ctrl+ALT+U. En la ventana para la configuración de las preferencias del usuario pulsamos sobre el botón *addons* y en la parte inferior de la ventana pulsamos el botón *install addons*. A continuación el asistente nos ofrecerá la posibilidad de elegir el *script* de python con el exportador de Blender a Ogre. Finalmente, el exportador aparecerá en el listado de la derecha y marcaremos la casilla de verificación.

Para comprobar que la instalación es correcta, nos dirigimos al menú *File >Export* y comprobamos que entre las opciones aparece *Ogre3D*.

11.2.2. Creación de un modelo en Blender

El objetivo es modelar una caja o cubo y aplicar una textura a cada una de las seis caras. Cuando ejecutamos Blender aparece en la escena un cubo creado por defecto, por tanto, no será necesario crear ningún modelo adicional para completar este ejercicio. En el caso de que quisiéramos añadir algún cubo más sería sencillo, bastaría con dirigirnos al menú *Add >Mesh >Cube* y aparecería un nuevo cubo en la escena.

Además del cubo, Blender crea por defecto una fuente de luz y una cámara. Al igual que sucede con los objetos, es posible añadir fuentes de luz adicionales y nuevas cámaras. Para este ejercicio será suficiente con los elementos creados por defecto.

11.2.3. Aplicación de texturas mediante UV Mapping

La técnica de texturizado UV Mapping permite establecer una correspondencia entre los vértices de un objeto y las coordenadas de una textura. Mediante esta técnica no es necesario crear un fichero o imagen para representar cada una de las texturas que se aplicará a cada una de las superficies que forman un objeto. En otras palabras, en un mismo fichero se puede representar la "piel" que cubrirá diferentes regiones de una superficie tridimensional. En está sección explicaremos como aplicar una textura a cada una de las seis caras del cubo creado en la sección anterior. La textura que se aplica a cada una de las caras puede aparecer desglosada en una misma imagen tal como muestra la Figura 11.10.

Figura 11.10: Orientación de las texturas aplicadas a un cubo

Cada uno de los recuadros corresponde a la textura que se aplicará a una de las caras del cubo en el eje indicado. En la Figura 11.11 se muestra la textura real que se aplicará al modelo; tal como se puede apreciar está organizada de la misma forma que se presenta en la Figura 11.10 (cargar la imagen llamada textura512.jpg que se aporta con el material del curso). Todas las texturas se encuentran en una misma imagen cuya resolución es 512x512. Los módulos de memoria de las tarjetas gráficas están

optimizados para trabajar con imágenes representadas en matrices cuadradas y con una altura o anchura potencia de dos. Por este motivo, la resolución de la imagen que contiene las texturas es de 512x512 a pesar de que haya espacios en blanco y se pueda compactar más reduciendo la altura.

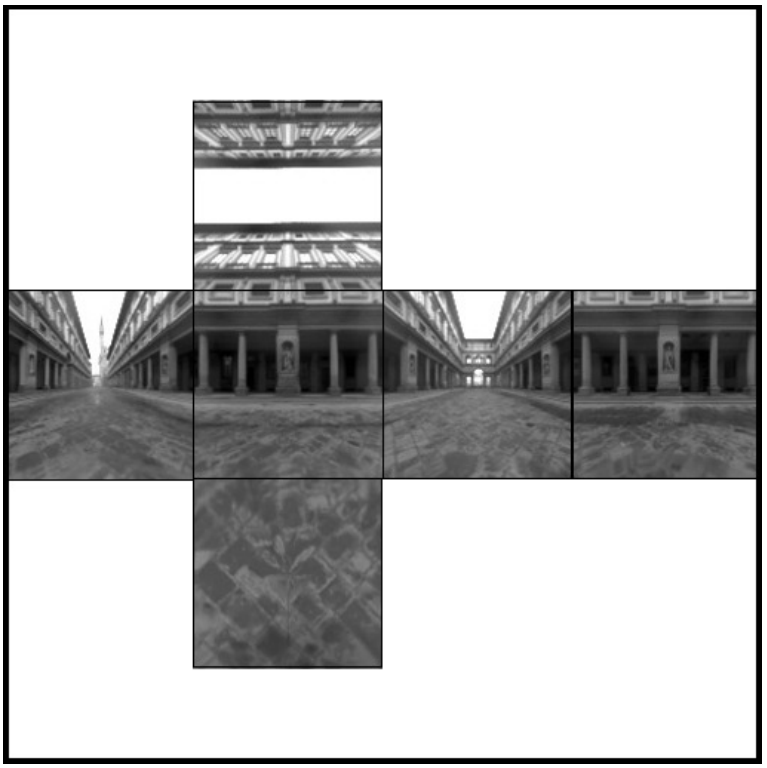


Figura 11.11: Imagen con resolución 512x512 que contiene las texturas que serán aplicadas a un cubo

Una vez conocida la textura a aplicar sobre el cubo y cómo está estructurada, se describirán los pasos necesarios para aplicar las texturas en las caras del cubo mediante la técnica UV Mapping. En primer lugar ejecutamos Blender y dividimos la pantalla en dos partes. Para ello situamos el puntero del ratón sobre el marco superior hasta que el cursor cambie de forma a una flecha doblemente punteada, pulsamos el botón derecho y elegimos en el menú flotante la opción *Split Area.*

En el marco de la derecha pulsamos el menú desplegable situado en la esquina inferior izquierda y elegimos la opción *UV/Image Editor*. A continuación cargamos la imagen de la textura (textura512.jpg); para ello pulsamos en el menú *Image >Open* y elegimos la imagen que contiene las texturas. Después de realizar estos datos la apariencia del editor de Blender debe ser similar a la que aparece en la Figura 11.12

En el marco de la izquierda se pueden visualizar las líneas que delimitan el cubo, coloreadas de color rosa, donde el modo de vista por defecto es *Wireframe*. Para visualizar por pantalla las texturas en todo momento elegimos el modo de vista *Textured* en el menú situado en el marco inferior *Viewport Shading*. Otra posibilidad consiste en pulsar las teclas [SHIFT] + [Z].

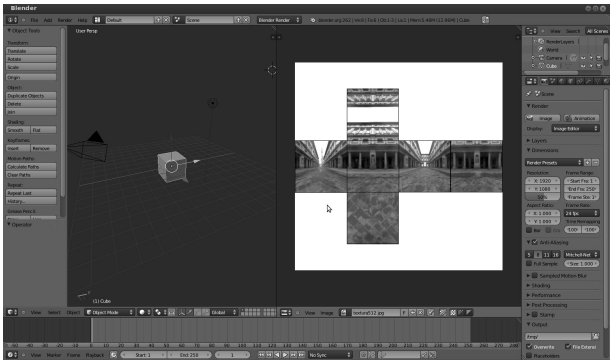


Figura 11.12: Editor de Blender con una división en dos marcos. En la parte izquierda aparece la figura geométrica a texturizar y en la parte derecha las texturas que serán aplicadas mediante la técnica de UV/Mapping

Por otro lado, el cubo situado en la escena tiene asociado un material por defecto, pero éste no tiene asignada ninguna textura (en el caso de que no tenga un material asociado, será necesario crear uno). Para ello, nos dirigimos al menú de materiales situado en la barra de herramientas de la derecha (ver Figura 11.13).

Figura 11.13: Selección del material del objeto

En segundo lugar, será necesario determinar la textura. Al panel de texturas se accede pulsando el icono situado a la derecha de el de materiales (ver Figura 11.14).

Figura 11.14: Definición de la textura del objeto

En el tipo de textura se elige *Image* y, más abajo, en el panel de *Image* pulsamos sobre el botón *Open* para abrir la imagen de la textura con formato jpg. Por otro lado, en el panel *Mapping*, será necesario elegir la opción *UV* en el menú *Generate*, para que las coordenadas de la textura se asocien correctamente al objeto en el proceso de renderizado.

El siguiente paso consiste en elegir los vértices de cada una de las caras del cubo y establecer una correspondencia con coordenadas de la textura. Para ello, pulsamos la tecla **TAB** para visualizar los vértices y nos aseguramos que todos están seleccionados (todos deben estar coloreados de amarillo). Para seleccionar o eliminar la selección de todos los vértices basta con pulsar la tecla **A**. Con todos los vértices del cubo seleccionados, pulsamos la tecla **U** y en el menú *UV Mapping* elegimos la opción *Cube Projection*. En el editor UV de la derecha deben aparecer tres recuadros nuevos de color naranja.

Pulsamos [A] para que ninguno de los vértices quede seleccionado y, de forma manual, seleccionamos los vértices de la cara superior (z+). Para seleccionar los vértices existen dos alternativas; la primera es seleccionar uno a uno manteniendo la tecla [SHIFT] pulsada y presionar el botón derecho del ratón en cada uno de los vértices. Una segunda opción es dibujar un área que encuadre los vértices, para ello hay que pulsar primero la tecla [B] (Pulsa el botón intermedio del ratón y muévelo para cambiar la perspectiva y poder asegurar así que se han elegido los vértices correctos).

En la parte derecha, en el editor UV se puede observar un cuadrado con los bordes de color amarillo y la superficie morada. Este recuadro corresponde con la cara del cubo seleccionada y con el que se pueden establecer correspondencias con coordenadas de la textura. El siguiente paso consistirá en adaptar la superficie del recuadro a la imagen que debe aparecer en la parte superior del cubo (z+), tal como indica la Figura 11.10. Para adaptar la superficie existen varias alternativas; una de ellas es escalar el cuadrado con la tecla [S] y para desplazarlo la tecla [G] (para hacer zoom sobre la textura se gira la rueda central del ratón). Un segunda opción es ajustar cada vértice de forma individual, para ello se selecciona un vértice con el botón derecho del ratón, pulsamos [G] y desplazamos el vértice a la posición deseada. Las dos alternativas descritas anteriormente no son excluyentes y se pueden combinar, es decir, se podría escalar el recuadro en primer lugar y luego ajustar los vértices, e incluso escalar de nuevo si fuera necesario.

Los pasos que se han realizo para establecer la textura de la parte superior del cubo deben ser repetidos para el resto de caras del cubo. Si renderizamos la escena pulsando [F12] podemos apreciar el resultado final.

Por último vamos a empaquetar el archivo *.blend* para que sea autocontenido. La textura es un recurso externo; si cargáramos el fichero *.blend* en otro ordenador posiblemente no se visualizaría debido a que las rutas no coinciden. Si empaquetamos el fichero eliminamos este tipo de problemas. Para ello, seleccionamos *File >External Data >Pack into .blend file*.

11.2.4. Exportación del objeto en formato Ogre XML

Para exportar la escena de Blender al formato de Ogre seleccionamos *File >Export >Ogre 3D*. Antes de exportar debemos asegurarnos de que nos encontramos en modo objeto y no en modo de edición, sino blender no nos permitirá exportar el objeto. En la parte izquierda aparecerá un nuevo panel con las opciones del exportador tal como muestra la Figura 11.15. Tal como se puede apreciar, el exportador dispone de múltiples opciones encuadradas en dos categorías: exportación de materiales y exportación de mallas.

Las opciones del exportador son las siguientes:

- Posibilidad de intercambiar los ejes de coordenadas
- **Separate Materials:** Exporta todos los materiales por separado, genera un archivo .material por cada uno de ellos, en lugar de aglomerar todos ellos en un único archivo.
- **Only Animated Bones:** Únicamente exporta aquellos huesos que han sido animados y forman parte de frames clave.
- **Export Scene:** Exportación de la escena actual.
- **Export Selected Only:** Exporta únicamente los objetos seleccionados.
- **Force Camera:** Exporta la cámara que esté activa en el momento actual.

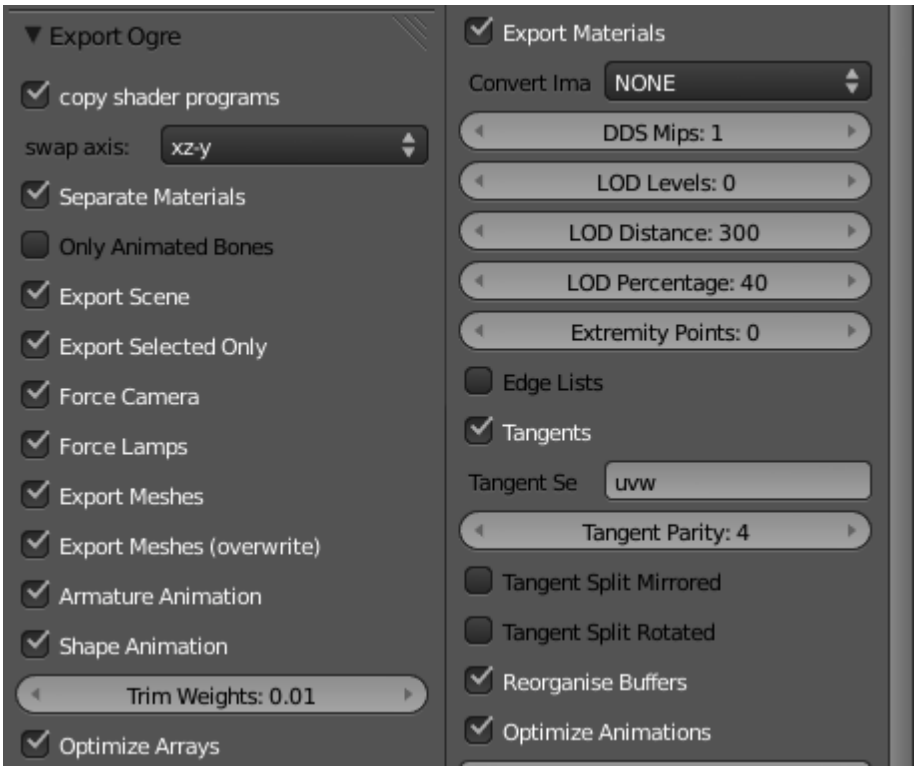


Figura 11.15: Exportador de Ogre integrado en Blender

- **Force Lamps:** Exportación de todas las luces de la escena.
- **Export Meshes:** Exportación de las mallas de los objetos.
- **Export Meshes (overwrite):** Exporta las mallas y las sobreescribe si se han creado con anterioridad.
- **Armature Animation:** Exportación del esqueleto del objeto y la animación de los huesos.
- **Shape Animation:** datos sobre la animación de personajes. Ofrece la posibilidad de ignorar los huesos cuyo valor esta por debajo del umbral definido en el parámetro *Trim Weights*.
- **Optimize Arrays:** optimiza el array de modificadores como instancias.
- **Export Materials:** Exportación de los materiales y generación de archivos .material
- Conversión de la imagen de la textura a otros formatos.
- Número de Mip Maps
- Número de niveles LOD en la malla
- Valor de incremento para reducir LOD
- Porcentaje de reducción LOD

- Generación de la lista con las aristas de la geometría del objeto
- Tangentes en la malla
- Reorganización de los buffers de vértices en la malla
- Optimización de las animaciones de la malla

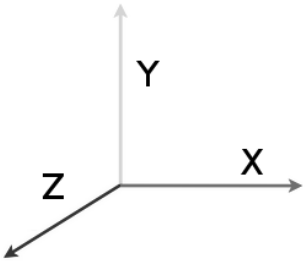


Figura 11.16: Sistema de coordenadas utilizado en Ogre.

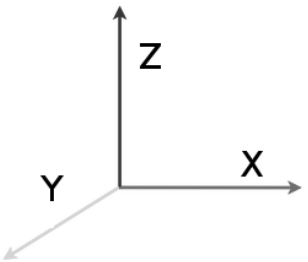


Figura 11.17: Sistema de coordenadas utilizado en Blender.

11.2.5. Carga del objeto en una aplicación Ogre

Para cargar el cubo exportado en Ogre vamos a partir de ejemplo visto al comienzo del curso *Hello World*. Recordemos que la estructura de directorios para la aplicación en Ogre era la siguiente:

- Directorios:
  - Include
  - Media
  - Obj
  - Plugins
  - src
- Ficheros en el directorio raíz del proyecto Ogre:
  - ogre.cfg
  - plugins.cfg
  - resources.cfg

Una vez que se ha exportado el cubo con las texturas asociadas y se han generado los ficheros *Cube.mesh*, *Cube.mesh.xml* y *Scene.material*, los incluimos en el directorio *media* (también incluimos la textura "textura512.jpg"). Los ficheros deben incluirse en este directorio porque así está configurado en el fichero *resources.cfg*, si deseáramos incluir los recursos en otro directorio deberíamos variar la configuración en dicho fichero.

En segundo lugar, cambiamos el nombre del ejecutable; para ello, abrimos el archivo *makefile* y en lugar de *EXEC := helloWorld*, ponemos *EXEC:= cubo*. Después es necesario cambiar el código de ejemplo para que cargue nuestro modelo. Abrimos el archivo */src/main.cpp* que se encuentra dentro en el directorio */src*. El código es el siguiente:

Listado 11.21: Código incluido en el archivo main.cpp

```
1  #include <ExampleApplication.h>
2  class SimpleExample : public ExampleApplication {
3      public : void createScene() {
4          Ogre::Entity *ent = mSceneMgr->createEntity("Sinbad", "
               Sinbad.mesh");
5          mSceneMgr->getRootSceneNode()->attachObject(ent);
6      }
7  };
8
9  int main(void) {
10     SimpleExample example;
11     example.go();
12     return 0;
13 }
```

En nuestro caso la entidad a crear es aquella cuya malla se llama *Cube.Mesh*, ésta ya contiene información sobre la geometría de la malla y las coordenadas UV en el caso de utilización de texturas.

Por lo tanto el código tras la modificación de la función *createScene*, sería el siguiente:

Listado 11.22: Modificación del fichero main.cpp para cargar el cubo modelado en Blender

```
1 public : void createScene() {
2     Ogre::Entity *ent = mSceneMgr->createEntity("Caja", "cubo.mesh"
          );
3     mSceneMgr->getRootSceneNode()->attachObject(ent);
4 }
```

Al compilar (make) y ejecutar (./Cubo) aceptamos las opciones que vienen por defecto y se puede observar el cubo pero muy lejano. A partir de ahora todas las operaciones que queramos hacer (translación, escalado, rotación, ...) tendrá que ser a través de código. A continuación se realiza una operación de traslación para acercar el cubo a la cámara:

Listado 11.23: Traslación del cubo para acercarlo hacia la cámara

```
1 Entity *ent1 = mSceneMgr->createEntity( "Cubo", "Cube.mesh" );
2 SceneNode *node1 = mSceneMgr->getRootSceneNode()->
      createChildSceneNode( );
3 node1->attachObject( ent1 );
4 node1->translate(Vector3(0, 0, 490 ));
```

Ejercicio: intentar escalar el cubo y rotarlo en función de los ejes usando *Pitch*, *Yaw* y *Roll*.

Listado 11.24: Ejemplo de escalado de un objeto

```
1 node1->scale( 3, 3, 3 );
```

Listado 11.25: Ejemplo de rotación

```
1 node1->yaw(Ogre::Degree( -90 ) );
2 node1->pitch(Ogre::Degree( -90 ) );
3 node1->roll(Ogre::Degree( -90 ) );
```

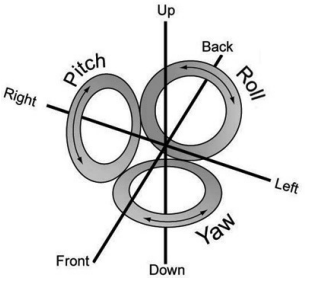


Figura 11.18: Operaciones de rotación.

11.3. Procesamiento de Recursos Gráficos

En esta sección estudiaremos cómo adaptar recursos 3D realizados con Blender en Ogre. Tendremos en cuenta aspectos relativos a la escala, posicionamiento de los objetos (respecto de su sistema de referencia local y global) y estudiaremos algunas herramientas disponibles en Blender para el posicionamiento preciso de modelos 3D.

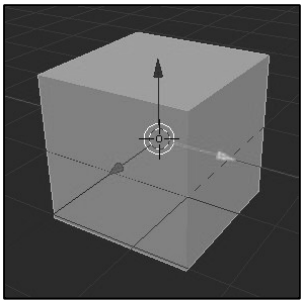


Figura 11.19: El centro del objeto define el origen del sistema de referencia local. Así, la posición de los vértices del cubo se definen según este sistema local.

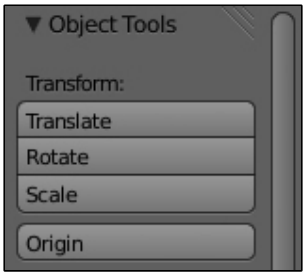


Figura 11.20: Botón *Origin* para elegir el centro del objeto, en el panel *Object Tools*.

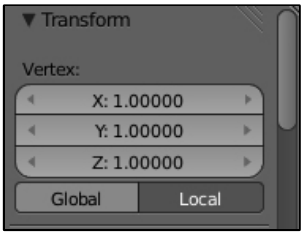


Figura 11.21: Propiedades del panel *Transform* en modo edición (tecla *T*). Permiten indicar las coordenadas de cada vértice con respecto del Sistema de Referencia Local y del Sistema de Referencia Universal (Global).

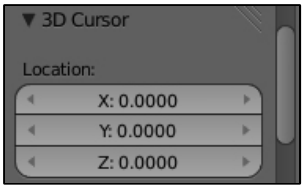


Figura 11.23: Posicionamiento numérico del cursor 3D.

Antes de continuar, el lector podría (opcionalmente) completar el estudio del capítulo 13, ya que utilizaremos como base el código fuente obtenido en el último ejemplo de dicho capítulo, y estudiaremos algunas cuestiones referentes al gestor de recursos.

Blender, a diferencia de los sistemas de CAD (Computer Aided Design) (que emplean modelos de CSG (Constructive Solid Geometry), es una herramienta de modelado de contorno B-REP (Boundary Representation). Esto implica que, a diferencia de los sistemas basados en CSG, trabaja con modelos huecos definidos por vértices, aristas y caras. Como hemos estudiado en el capítulo de introducción matemática, las coordenadas de estos modelos se especifican de forma relativa a su centro, que define el origen de su sistema de referencia local.

En Blender el centro del objeto se representa mediante un punto de color rosa (ver Figura 11.19). Si tenemos activos los manejadores en la cabecera de la ventana 3D , será el punto de donde comienzan estos elementos de la interfaz.

Este centro define el sistema de referencia local, por lo que resulta crítico poder cambiar el centro del objeto ya que, tras su exportación, se dibujará a partir de esta posición.

Con el objeto seleccionado en Modo de Objeto se puede indicar a Blender que recalcule el centro geométrico del objeto en el panel de *Object Tools* (accesible con la tecla T), en el botón Origin (ver Figura 11.20). Una vez pulsado *Origin*, podemos elegir entre **Geometry to Origin**, que desplaza los vértices del objeto de modo que se ajustan a la posición fijada del centro geométrico, **Origin to Geometry** que realiza la misma operación de alineación, pero desplazando el centro en lugar de mover los vértices, o **Origin to 3D Cursor** que cambia la posición del centro del objeto a la localización actual del cursor 3D.

En muchas ocasiones, es necesario situar el centro y los vértices de un modelo con absoluta precisión. Para realizar esta operación, basta con pulsar la tecla N, y aparecerá el panel de Transformación *Transform* a la derecha de la vista 3D (ver Figura 11.21), en el que podremos especificar las coordenadas específicas del centro del objeto. Si estamos en modo edición, podremos indicar las coordenadas a nivel de vértice. Es interesante que esas coordenadas se especifiquen *localmente* (botón Local activado), porque el exportador de Ogre trabajará con coordenadas locales relativas a ese centro.

El centro del objeto puede igualmente situarse de forma precisa, empleando la posición del puntero 3D. El puntero 3D puedes situarse en cualquier posición del espacio con precisión. En el panel *Transform* comentado anteriormente (accesible mediante la tecla T) pueden especificarse numéricamente las coordenadas 3D (ver Figura 11.23). Posteriormente, utilizando el botón Origin (ver Figura 11.20), y eligiendo la opción *Origin to 3D Cursor* podremos situar el centro del objeto en la posición del puntero 3D.

Es muy importante que las transformaciones de modelado se apliquen finalmente a las coordenadas locales del objeto, para que se apliquen de forma efectiva a las coordenadas de los vértices. Es decir, si después de trabajar con el modelo, pulsando la tecla N en modo Objeto, el valor *Scale* es distinto de 1 en alguno de los ejes, implica que esa transformación se ha realizado en modo objeto, por lo que los vértices del mismo no tendrán esas coordenadas asociadas. De igual modo, también hay que prestar atención a la rotación del objeto (estudiaremos cómo resolver este caso más adelante).

La Figura 11.22 muestra un ejemplo de este problema; el cubo de la izquierda se ha escalado en modo edición, mientras que el de la derecha se ha escalado en modo objeto. El modelo de la izquierda se exportará correctamente, porque los vértices tienen asociadas sus coordenadas locales. El modelo de la derecha sin embargo aplica una transformación a nivel de objeto, y será exportado como un cubo.

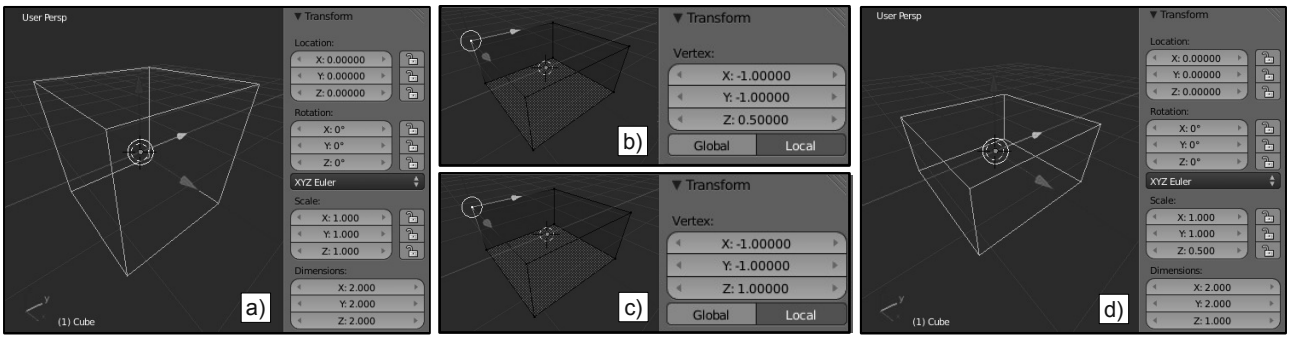


Figura 11.22: Aplicación de transformaciones geométricas en modo Edición. **a)** Situación inicial del modelo al que le aplicaremos un escalado de 0.5 respecto del eje Z. **b)** El escalado se aplica en modo Edición. El campo Vertex Z muestra el valor correcto. **c)** Si el escalado se aplicó en modo objeto, el campo Vertex Z no refleja la geometría *real* del objeto. Esto puede comprobarse fácilmente si alguno de los campos *Scale* no son igual a uno, como en **d)**.

Blender dispone de una orden para aplicar las transformaciones realizadas en modo Objeto al sistema de coordenadas local del objeto. Para ello, bastará con pulsar **Control A** y elegir *Apply/ Rotation & Scale*, o bien desde la cabecera de la ventana 3D en *Object/ Apply/ Rotation & Scale*.

Otra operación muy interesante consiste en posicionar un objeto con total precisión. Para realizar esta operación puede ser suficiente con situarlo numéricamente, como hemos visto anteriormente, accediendo al panel de transformación mediante la tecla N. Sin embargo, otras veces necesitamos situarlo en una posición relativa a otro objeto; necesitaremos situarlo empleando el cursor 3D.

El centro de un objeto puede situarse en la posición del puntero 3D, y el puntero 3D puede situarse en cualquier posición del espacio. Podemos, por ejemplo, en modo edición situar el puntero 3D en la posición de un vértice de un modelo, y situar ahí el centro del objeto. Desde ese momento, los desplazamientos del modelo se harán tomando como referencia ese punto.

Mediante el atajo Shift S *Cursor to Selected* podemos situar el puntero 3D en la posición de un elemento seleccionado (por ejemplo, un vértice, o en el centro de otro objeto que haya sido seleccionado en modo objeto) y mediante Shift S *Selection to Cursor* moveremos el objeto seleccionado a la posición del puntero 3D (ver Figura 11.24). Mediante este sencillo mecanismo de 2 pasos podemos situar los objetos con precisión, y modificar el centro del objeto a cualquier punto de interés.

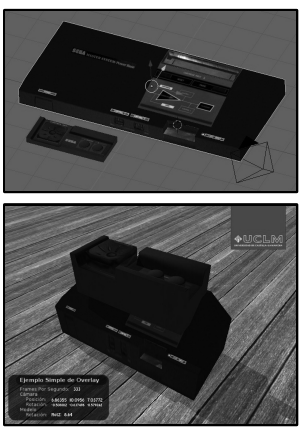


Figura 11.25: Resultado de exportar directamente el modelo de la *Master System* desde Blender. La imagen superior muestra el modelo en Blender, y las inferior el resultado de desplegar el modelo en Ogre, con proporciones claramente erróneas.

11.3.1. Ejemplo de uso

A continuación veremos un ejemplo de utilización de los operadores comentados anteriormente. Partiremos de un sencillo modelo (de 308 vértices y 596 caras triangulares) creado en Blender, que se muestra en la Figura 11.25. Al importar el modelo en Ogre, se ha creado un plano manualmente que sirve como "*base*", y que está posicionado en $Y = 0$ (con vector normal el Y unitario). El código 11.26 muestra las líneas más relevantes relativas a la creación de la escena.

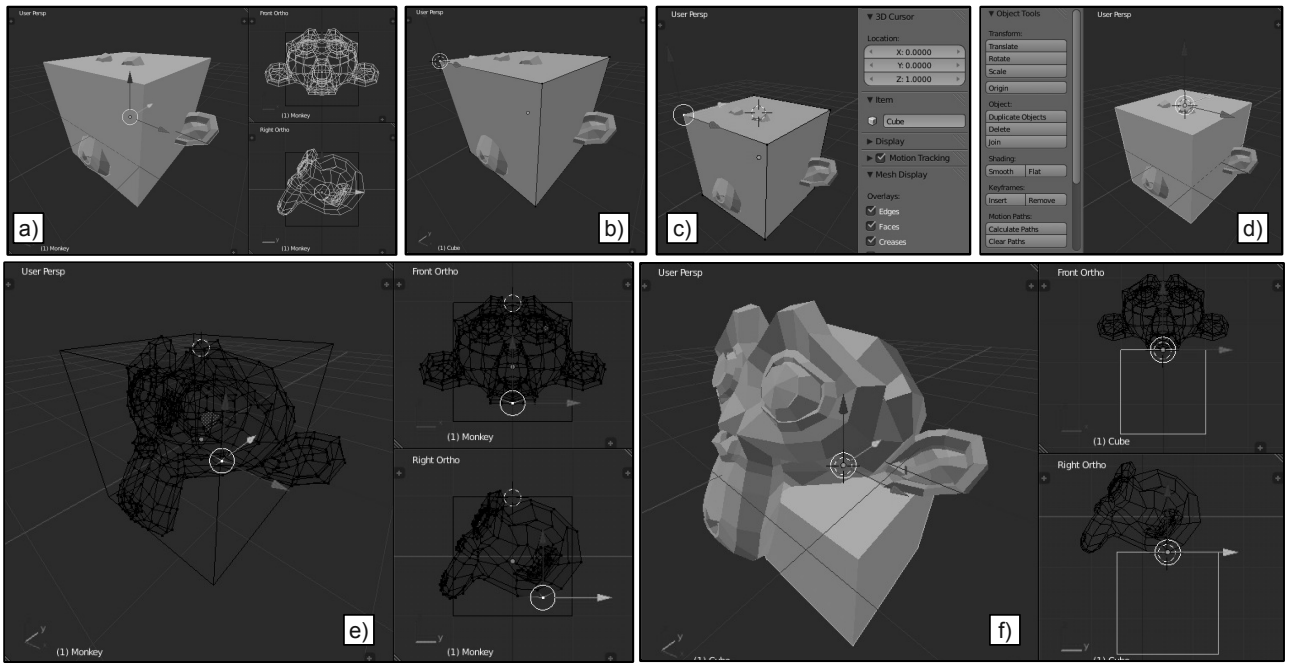


Figura 11.24: Ejemplo de uso del operador *Cursor to Selected* y *Selection to Cursor*. **a)** Posición inicial de los dos modelos. Queremos llegar a la posición *f)*. En modo edición, elegimos un vértice superior del cubo y posicionamos el puntero 3D ahí (empleando *Shift S Cursor to Selected*). **c)** A continuación modificamos el valor de X e Y de la posición del cursor numéricamente, para que se sitúe en el centro de la cara. **d)** Elegimos *Origin to 3D Cursor* entre las opciones disponibles en el botón *Origin* del panel *Object Tools*. Ahora el cubo tiene su centro en el punto medio de la cara superior. **e)** Elegimos el vértice señalado en la cabeza del mono y posicionamos ahí el puntero 3D. Finalmente en **f)** cambiamos la posición del cubo de forma que interseca exactamente en ese vértice (*Shift S/ Selection to Cursor*).

El problema viene asociado a que el modelo ha sido construido empleando transformaciones a nivel de objeto. Si accedemos a las propiedades de transformación [N] para cada objeto, obtenemos la información mostrada en la Figura 11.26. Como vemos, la escala a nivel de objeto de ambos modelos es distinta de 1.0 para alguno de los ejes, lo que indica que la transformación se realizó a nivel de objeto.

Figura 11.26: Propiedades de transformación de los dos objetos del ejemplo.

Aplicaremos la escala (y la rotación, aunque en este caso el objeto no fue rotado) a los vértices del modelo, eliminando cualquier operación realizada en modo objeto. Para ello, con cada objeto seleccionado, pulsaremos [Control] [A] *Apply / Rotation & Scale*. Ahora la escala (ver Figura 11.26) debe mostrar un valor de 1.0 en cada eje.

Como hemos visto, la elección correcta del centro del objeto facilita el código en etapas posteriores. Si el modelo está normalizado (con escala 1.0 en todos los ejes), las coordenadas del espacio 3D de Blender pueden ser fácilmente transformadas a coordenadas de Ogre. En este caso, vamos a situar el centro de cada objeto de la escena de forma que esté situado exactamente en el $Z = 0$. Para ello, con cada objeto seleccionado, pulsaremos en [Origin] *Origin to 3D Cursor* (del panel *Object Tools*).

Ahora posicionaremos el cursor 3D en esa posición [Shift] [S] *Cursor to Selected* y modificaremos numéricamente la coordenada del cursor 3D, para que se sitúe en $Z = 0$ (accediendo al panel de *Propiedades* [N]). El resultado de posicionar el cursor 3D se muestra en la Figura 11.27.

Listado 11.26: Fragmento de MyApp.c

```
int MyApp::start() {
  // ... Carga de configuracion, creacion de window ...
  Ogre::Camera* cam = _sceneManager->createCamera("MainCamera");
  cam->setPosition(Ogre::Vector3(5,20,20));
  cam->lookAt(Ogre::Vector3(0,0,0));
  cam->setNearClipDistance(5);
  cam->setFarClipDistance(100);
  // ... Viewport, y createScene...
}

void MyApp::createScene() {
  Ogre::Entity* ent1 = _sceneManager->createEntity("MS.mesh");
  Ogre::SceneNode* node1 = _sceneManager->createSceneNode("MS");
  node1->attachObject(ent1);
  node1->translate(0,2,0);
  _sceneManager->getRootSceneNode()->addChild(node1);

  Ogre::Entity* ent2 = _sceneManager->createEntity("Mando.mesh");
  Ogre::SceneNode* node2 = _sceneManager->createSceneNode("Mando");
  node2->attachObject(ent2);
  node2->translate(0,2,0);
  node1->addChild(node2);
  // ... creacion del plano, luces, etc...
}
```

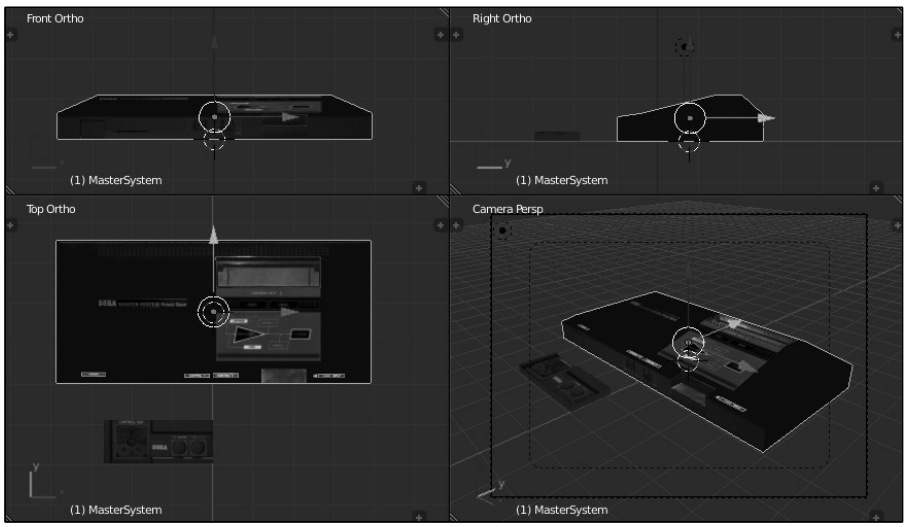


Figura 11.27: Posicionamiento del puntero 3D en relación al centro del objeto. En muchas situaciones puede ser conveniente especificar que varios objetos tengan el centro en una misma posición del espacio (como veremos en la siguiente sección). Esto puede facilitar la construcción de la aplicación.

Sistemas de Coordenadas: Recordemos que los sistemas de coordenadas de Ogre y Blender siguen convenios diferentes. Si utilizamos la opición del exportador de Ogre *Swap axis* por defecto (con valor *xz-y*), para obtener las coordenadas equivalentes de Ogre desde Blender bastará con aplicar la misma coordenada X, la coordenada Z de Blender utilizarla en Y en Ogre, y la coordenada Y de Blender aplicarla invertida en Z en Ogre.

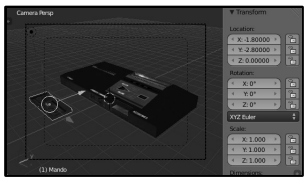


Figura 11.28: Posiconamiento del objeto mando.

Ahora podemos exportar el modelo a Ogre, y las proporciones se mantendrán correctamente. Sería conveniente posicionar exactamente el mando en relación a la consola, tal y como está en el modelo .blend. Para ello, podemos modificar y anotar manualmente la posición de objeto 3D (en relación con el SRU). Como el centro de la consola se ha posicionado en el origen del SRU, la posición del centro del mando será relativa al centro de la consola. Bastará con consultar estos valores (*Location X, Y, Z*) en el panel *Transform* (ver Figura 11.28) y utilizarlos en Ogre.

En el ejemplo de la Figura 11.28, el objeto tiene asociadas en Blender las coordenadas (-1.8, -2.8, 0). Al aplicar la equivalencia aplicada por el exportador (por defecto), obtendríamos el equivalente en Ogre de (-1.8, 0, 2.8). De este modo, el siguiente fragmento de código muestra la creación de la escena correcta en Ogre. En este caso ya no es necesario aplicar ninguna traslación al objeto *MS* (se posicionará en el origen del SRU). Por su parte, al objeto *Mando* se aplicará la traslación indicada en la línea [9], que se corresponde con la obtenida de Blender.

Automatízate!!

Obviamente, el proceso de exportación de los datos relativos al posicionamiento de objetos en la escena (junto con otros datos de interés) deberán ser automatizados definiendo un formato de escena para Ogre. Este formato tendrá la información necesaria para cada juego particular.

Listado 11.27: Fragmento de MyApp.c

```
1  Ogre::Entity* ent1 = _sceneManager->createEntity("MS.mesh");
2  Ogre::SceneNode* node1 = _sceneManager->createSceneNode("MS");
3  node1->attachObject(ent1);
4  _sceneManager->getRootSceneNode()->addChild(node1);
5
6  Ogre::Entity* ent2 = _sceneManager->createEntity("Mando.mesh");
7  Ogre::SceneNode* node2 = _sceneManager->createSceneNode("Mando");
8  node2->attachObject(ent2);
9  node2->translate(-1.8,0,2.8);
10 node1->addChild(node2);
```

Por último, sería deseable poder exportar la posición de las cámara para percibir la escena con la misma configuración que en Blender. Existen varias formas de configurar el *camera pose*. Una de las más cómodas para el diseñador es trabajar con la posición de la cámara y el punto al que ésta mira. Esta especificación es equivalente a indicar la posición y el punto *look at* (en el primer fragmento de código de la sección, en las líneas [4] y [5].

Para que la cámara apunte hacia un objeto de la escena, puede crearse una restricción de tipo *Track To*. Para ello, añadimos un objeto vacío a la escena (que no tiene representación), mediante [Shift] [A] *Add / Empty*. Ahora seleccionamos primero la cámara, y luego con [Shift] pulsado seleccionamos el Empty y pulsamos [Control] [T] *Track To Constraint*. De este modo, si desplazamos la cámara, obtendremos que siempre está *mirando* hacia el objeto Empty creado (ver Figura 11.29).

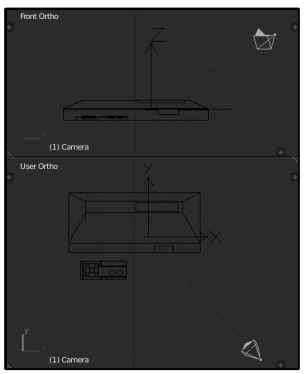


Figura 11.29: Track de la cámara al objeto Empty. Blender representa la relación con una línea punteada que va de la cámara al objeto Empty.

Cuando tengamos la "fotografía" de la escena montada, bastará con consultar el valor de posición de la cámara y el *Empty*, y asignar esos valores al código de posicionamiento y *look at* de la misma (teniendo en cuenta las diferencias entre sistemas de coordenadas de Blender y Ogre). El resultado se encuentra resumido en la Figura 11.30.

11.4. Gestión de Recursos y Escena

En esta sección estudiaremos un ejemplo que cubre varios aspectos que no han sido tratados en el documento, relativos al uso del gestor de recursos y de escena. Desarrollaremos un demostrador de *3D Picking* que gestionará manualmente el puntero del ratón (mediante *overlays*), gestionará recursos empaquetados en archivos `.zip` (empleando las facilidades que proporciona Ogre), cargará geometría estática y utilizará el potente sistema de *queries* del *SceneManager* con máscaras.

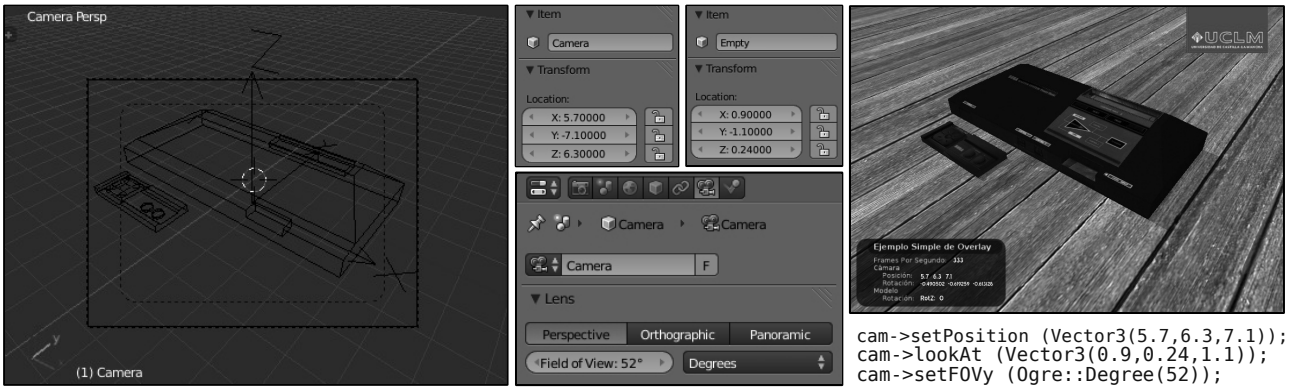


Figura 11.30: Resultado de aplicar la misma cámara en Blender y Ogre. El campo de visión de la lente puede consultarse en las propiedades específicas de la cámara, indicando que el valor quiere representarse en grados (ver lista desplegable de la derecha)

La Figura 11.33 muestra el interfaz del ejemplo desarrollado. Mediante la rueda del ratón es posible desplazar la cámara. Si se pulsa la rueda, se aplicará un rotación sobre la misma. Si se pulsa el botón izquierdo del ratón sobre un objeto de colisión del escenario (como el mostrado en la figura), se seleccionará mostrando su caja límite *bounding box*. Si se pulsa con el botón izquierdo sobre el suelo de la sala, se añadirá aleatoriamente una caja de tipo 1 o 2. Si pinchamos con el botón izquierdo sobre alguna de las cajas creadas, se seleccionará. Mediante el botón derecho únicamente podremos seleccionar cajas creadas (no se añadirán nunca cajas, aunque pulsemos sobre el suelo de la sala).

Sobre una caja creada, podemos aplicar tres operaciones: con la tecla Supr eliminamos el objeto. Con la tecla R rotaremos la caja respecto de su eje Y. Finalmente, con la tecla S modificamos su escala. El operador de escala y de rotación permiten invertir su comportamiento si pulsamos simultáneamente la tecla Shift. Veamos a continuación algunos aspectos relevantes en la construcción de este ejemplo.

11.4.1. Recursos empaquetados

El gestor de recursos de Ogre permite utilizar ficheros `.zip` definiendo en el archivo de configuración de recursos que el tipo es *Zip*.

De este modo, el archivo de configuración de recursos asociado a este ejemplo se muestra en la Figura 11.31. La implementación del cargador de recursos que estudiamos en el capítulo 13 permite su utilización directamente.

```
[General]
FileSystem=media
Zip=media/cube.zip
Zip=media/stage.zip
Zip=media/colision.zip
Zip=media/overlay.zip
```

Figura 11.31: Contenido del archivo de configuración de recursos *resource.cfg*.

11.4.2. Gestión del ratón

En el siguiente listado se resume el código necesario para posicionar una imagen usada como puntero del ratón en Ogre. La Figura 11.32 define el material creado (textura con transparencia) para cargar la imagen que emplearemos como puntero.

```
material pointer{
 technique {
  pass {
   scene_blend src_alpha
     one_minus_src_alpha
   texture_unit {
     texture cursor.png
}}}}
```

Figura 11.32: Material asociado al puntero del ratón.

En la línea 11 se posiciona el elemento del overlay llamado *cursor* en la posición absoluta obtenida del ratón por OIS (líneas 1 y 2). En el constructor del *FrameListener* hay que indicar a OIS las dimensiones de la ventana, para que pueda posicionar adecuadamente el ratón (de otra forma, trabajará en un cuadrado de 50 píxeles de ancho y alto). Esta operación se realiza como se indica en las líneas 14–15.

Figura 11.33: Ejemplo de aplicación que desarrollaremos en esta sección. El interfaz permite seleccionar objetos en el espacio 3D y modificar algunas de sus propiedades (escala y rotación).

El desplazamiento de la rueda del ratón se obtiene en la coordenada Z con *getMouseState* (línea 5). Utilizamos esta información para desplazar la cámara relativamente en su eje local Z.

Listado 11.28: Fragmentos de MyFrameListener.cpp

```
1  int posx = _mouse->getMouseState().X.abs;   // Posicion del puntero
2  int posy = _mouse->getMouseState().Y.abs;   // en pixeles.
3
4  // Si usamos la rueda, desplazamos en Z la camara -----------
5  vt+= Vector3(0,0,-10)*deltaT * _mouse->getMouseState().Z.rel;
6  _camera->moveRelative(vt * deltaT * tSpeed);
7
8  // Gestion del overlay ----------------------------
9  OverlayElement *oe;
10 oe = _overlayManager->getOverlayElement("cursor");
11 oe->setLeft(posx);  oe->setTop(posy);
12
13 // En el constructor de MyFrameListener...
14 _mouse->getMouseState().width = _win->getWidth();
15 _mouse->getMouseState().height = _win->getHeight();
```

11.4.3. Geometría Estática

Como hemos estudiado anteriormente, el modelo abstracto de los *MovableObject* abarca multitud de tipos de objetos en la escena (desde luces, cámaras, entidades, etc...). Uno de los tipos más empleados son las mallas poligionales, que tienen asociada información geométrica y datos específicos para el posterior uso de materiales.

La **geometría estática**, como su nombre indica, está pensada para elementos gráficos que no modifican su posición durante la ejecución de la aplicación. Está pensada para enviar *pocos* paquetes (lotes) *grandes* de geometría a la GPU en lugar de muchos pequeños. Esto permite optimizar el rendimiento en su posterior despliegue. De este modo, a menor número de paquetes enviados a la GPU tendremos mayor rendimiento en la aplicación.

Sobre eficiencia...

Realizar 100 llamadas de 10.000 polígonos a la GPU puede ser un orden de magnitud menos eficiente que realizar 10 llamadas de 100.000 polígonos.

El uso de la memoria empleando *geometría estática* sin embargo es mayor. Mientras que los *MovableObject* comparten mallas poligonales (siempre que sean instancias del mismo objeto), empleando geometría estática se copian los datos para cada instancia de la geometría.

Existen algunas características principales que deben tenerse en cuenta a la hora de trabajar con geometría estática:

- **Construcción**. La geometría estática debe ser construída previamente a su uso. Esta etapa debe realizarse una única vez, por lo que puede generarse en la carga del sistema y no supone una carga real en la tasa de refresco interactiva del juego.
- **Gestión de materiales**. El número de materiales diferentes asociados a la geometría estática delimita el número de lotes (paquetes) que se enviarán a la GPU. De este modo, aunque un bloque de geometría estática contenga gran cantidad de polígonos, se crearán tantos paquetes como diferentes materiales tenga asociado el bloque. Así, existe un compromiso de eficiencia relacionado con el número de materiales diferentes asociados a cada paquete de geometría estática.
- **Rendering en grupo**. Aunque únicamente una pequeña parte de la geometría estática sea visible en el *Frustum* todo el paquete será enviado a la GPU para su representación. De este modo, es conveniente separar la geometría estática en diferentes bloques para evitar el despliegue global de todos los elementos.

En el siguiente fragmento de código relativo a MyApp.c se muestra las instrucciones necesarias para definir un paquete de geometría estática en Ogre. Basta con llamar al *SceneManager*, e indicarle el nombre del bloque a definir (en este caso "SG" en la línea [1]). A continuación se añade una entidad cargada desde un archivo `.mesh`. Finalmente en la línea [4] se ejecuta la operación de construcción del bloque de geometría.

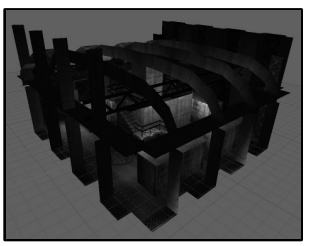


Figura 11.34: El escenario del ejemplo está realizado en baja poligonalización (1616 vértices y 1003 caras). Aun así, el uso de geometría estática *acelera* su despliegue en 3x!.

Listado 11.29: Fragmento de MyApp.c

```
1 StaticGeometry* stage = _sceneManager->createStaticGeometry("SG");
2 Entity* ent1 = _sceneManager->createEntity("Escenario.mesh");
3 stage->addEntity(ent1, Vector3(0,0,0));
4 stage->build();  // Operacion para construir la geometria
```

Otro aspecto importante a tener en cuenta a la hora de trabajar con geometría estática es que no es tenida en cuenta a la hora de realizar preguntas al gestor de escena. En la siguiente sección veremos cómo *resolver* este inconveniente.

11.4.4. Queries

El gestor de escena permite resolver preguntas relativas a la relación espacial entre los objetos de la escena. Estas preguntas pueden plantearse relativas a los objetos móviles *MovableObject* y a la geometría del mundo *WorldFragment*.

Ogre no gestiona las preguntas relativas a la geometría estática. Una posible solución pasa por crear objetos móviles invisibles de baja poligonalización que servirán para realizar estas preguntas. Estos objetos están exportados conservando las mismas dimensiones y rotaciones que el bloque de geometría estática (ver Figura 11.35). Además, para evitar tener que cargar manualmente los centros de cada objeto, todos los bloques han redefinido su centro en el origen del SRU (que coincide con el centro de la geometría estática). El siguiente listado muestra la carga de estos elementos en el grafo de escena.

Listado 11.30: Fragmento de MyApp.cpp (Create Scene)

```
1 // Objeto movable "suelo"para consultar al SceneManager
```

```
 2 SceneNode *nodecol = _sceneManager->createSceneNode("Col_Suelo");
 3 Entity *entcol = _sceneManager->createEntity
 4                                  ("Col_Suelo", "Col_Suelo.mesh");
 5 entcol->setQueryFlags(STAGE);   // Usamos flags propios!
 6 nodecol->attachObject(entcol);
 7 nodecol->setVisible(false);     // Objeto oculto
 8 _sceneManager->getRootSceneNode()->addChild(nodecol);
 9
10 // Cajas del escenario (baja poligonalizacion)
11 stringstream sauxnode, sauxmesh;
12 string s = "Col_Box";
13 for (int i=1; i<6; i++) {
14   sauxnode << s << i; sauxmesh << s << i << ".mesh";
15   SceneNode *nodebox = _sceneManager->createSceneNode
16                                                (sauxnode.str());
17   Entity *entboxcol = _sceneManager->createEntity
18                                (sauxnode.str(), sauxmesh.str());
19   entboxcol->setQueryFlags(STAGE);    // Escenario
20   nodebox->attachObject(entboxcol);
21   nodebox->setVisible(false);
22   nodecol->addChild(nodebox);
23   sauxnode.str(""); sauxmesh.str(""); // Limpiamos el stream
24 }
```

Cabe destacar el establecimiento de flags propios (en las líneas 5 y 19) que estudiaremos en la sección 11.4.5, así como la propiedad de visibilidad del nodo (en 7 y 21). El bucle definido en 13-23 sirve para cargar las cajas límite mostradas en la Figura 11.35.

Las preguntas que pueden realizarse al gestor de escena se dividen en cuatro categorías principales: **1. Caja límite**, definida mediante dos vértices opuestos, **2. Esfera**, definida por una posición y un radio, **3. Volumen**, definido por un conjunto de tres o más planos y **4. Rayo**, denifido por un punto (origen) y una dirección.

Otras intersecciones

Ogre soporta igualmente realizar consultas sobre cualquier tipo de intersecciones arbitrarias entre objetos de la escena.

En este ejemplo utilizaremos las intersecciones Rayo-Plano. Una vez creado el objeto de tipo *RaySceneQuery* en el constructor (mediante una llamada a *createRayQuery* del *SceneManager*), que posteriormente será eliminado en el destructor (mediante una llamada a *destroyQuery* del *SceneManager*), podemos utilizar el objeto para realizar consultas a la escena.

En la línea 20 se utiliza un método auxiliar para crear la Query, indicando las coordenadas del ratón. Empleando la función de utilidad de la línea 2, Ogre crea un rayo con origen en la cámara y la dirección indicada por las dos coordendas X, Y normalizadas (entre 0.0 y 1.0) que recibe como argumento (ver Figura 11.36). En la línea 4 se establece ese rayo para la consulta y se indica en la línea 5 que ordene los resultados por distancia de intersección. La consulta de tipo rayo devuelve una lista con todos los objetos que han intersecado con el rayo[1]. Ordenando el resultado por distancia, tendremos como primer elemento el primer objeto con el que ha *chocado* el rayo. La ejecución de la consulta se realiza en la línea 21.

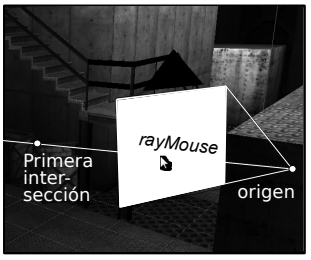


Figura 11.36: Empleando la llamada a *getCameraToViewportRay*, el usuario puede obtener un rayo con origen en la posición de la cámara, y con la dirección definida por la posición del ratón.

Para recuperar los datos de la consulta, se emplea un iterador. En el caso de esta aplicación, sólo nos interesa el primer elemento devuelto por la consulta (el primer punto de intersección), por lo que en el *if* de la línea 25 preguntamos si hay algún elemento devuelto por la consulta.

[1] El rayo se describe por un punto de origen y una dirección. Desde ese origen, y siguiendo esa dirección, el rayo describe una línea infinita que podrá intersecar con *infinitos* objetos

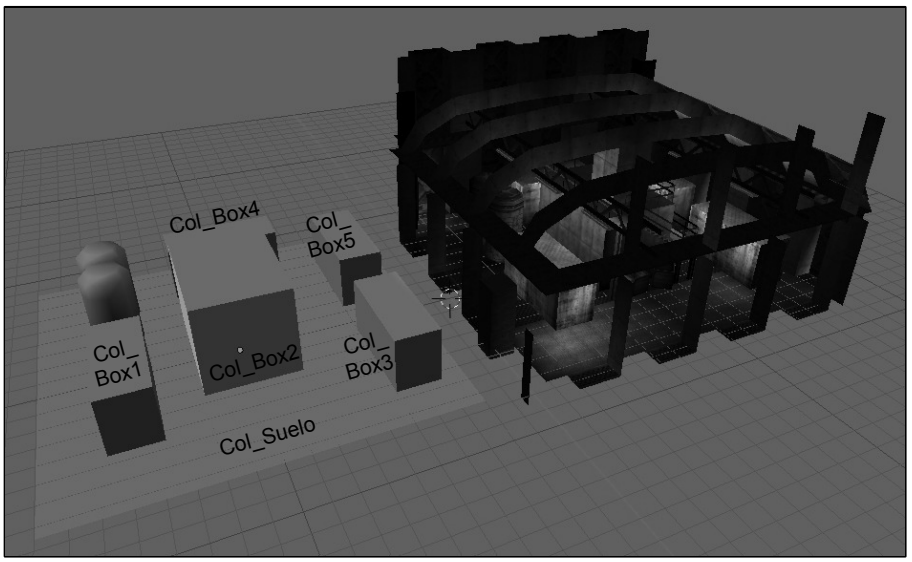


Figura 11.35: A la izquierda la geometría utilizada para las preguntas al gestor de escena. A la derecha la geometría estática. El centro de todas las entidades de la izquierda coinciden con el centro del modelo de la derecha. Aunque en la imagen los modelos están *desplazados* con respecto del eje X, en realidad ambos están exactamente en la misma posición del mundo.

La creación de las *Queries* son costosas computacionalmente (por lo que interesa realizarlas al inicio; en los constructores). Sin embargo su ejecución puede realizarse sin problema en el bucle principal de dibujado. El rendimiento se verá notablemente afectado si creas y destruyes la *query* en cada frame.

El iterador obtiene punteros a objetos de tipo *RaySceneQueryResultEntry*, que cuentan con 3 atributos públicos. El primero llamado distance es de tipo *Real* (usado en la línea 35) nos indica la distancia de intersección desde el origen del rayo. El segundo atributo llamado *movable* contiene un puntero al MovableObject con el que intersecó (si existe). El tercer atributo *worldFragment* contiene un puntero al objeto de ese tipo (en este ejemplo no utilizamos geometría de esta clase).

El test de intersección rayo-objeto se realiza empleando cajas límite. Esto resulta muy eficiente (ya que únicamente hay que comparar con una caja y no con todos los polígonos que forman la entidad), pero tiene el inconveniente de la pérdida de precisión (Figura 11.37).

Finalmente, para añadir una caja en el punto de intersección del rayo con el suelo, tenemos que obtener la posición en el espacio 3D. Ogre proporciona una función de utilidad asociada a los rayos, que permite obtener el punto 3D, indicando una distancia desde el origen (línea 35). De esta forma, indicando la distancia de intersección (obtenida en el iterador) en el método *getPoint* del rayo, obtenemos el *Vector3* que usamos directamente para posicionar el nuevo nodo en la escena (ver listado 11.31).

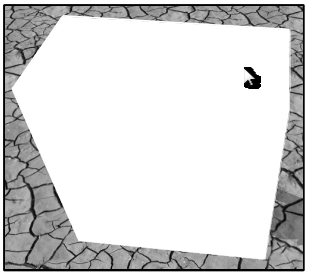


Figura 11.37: En el caso de objetos móviles, Ogre emplea una caja límite (definida por las coordenadas mayor y menor de los vértices del modelo) para calcular la intersección *rayo-objeto* y optimizar así los cálculos. Esto implica que, en el caso del test de intersección de la figura daría que hay colisión entre el rayo y el objeto. No obstante, es posible realizar *manualmente* un test de colisión con precisión (a nivel de polígono) si el juego lo requiere.

Listado 11.31: Fragmento de MyFrameListener.cpp

```
1  Ray MyFrameListener::setRayQuery(int posx, int posy, uint32 mask) {
2    Ray rayMouse = _camera->getCameraToViewportRay
3      (posx/float(_win->getWidth()), posy/float(_win->getHeight()));
4    _raySceneQuery->setRay(rayMouse);
5    _raySceneQuery->setSortByDistance(true);
6    _raySceneQuery->setQueryMask(mask);
7    return (rayMouse);
8  }
9
10 bool MyFrameListener::frameStarted(const FrameEvent& evt) {
11   // ... Codigo anterior eliminado...
12   if (mbleft || mbright) {  // Boton izquierdo o derecho ------
13     if (mbleft) mask = STAGE | CUBE1 | CUBE2;  // Todos
14     if (mbright) mask = ~STAGE;   // Todo menos el escenario
15
16     if (_selectedNode != NULL) {  // Si hay alguno seleccionado...
17       _selectedNode->showBoundingBox(false);  _selectedNode = NULL;
18     }
19
20     Ray r = setRayQuery(posx, posy, mask);
21     RaySceneQueryResult &result = _raySceneQuery->execute();
22     RaySceneQueryResult::iterator it;
23     it = result.begin();
24
25     if (it != result.end()) {
26       if (mbleft) {
27           if (it->movable->getParentSceneNode()->getName() ==
28                                                    "Col_Suelo") {
29             SceneNode *nodeaux = _sceneManager->createSceneNode();
30             int i = rand()%2;   stringstream saux;
31             saux << "Cube" << i+1 << ".mesh";
32             Entity *entaux=_sceneManager->createEntity(saux.str());
33             entaux->setQueryFlags(i?CUBE1:CUBE2);
34             nodeaux->attachObject(entaux);
35             nodeaux->translate(r.getPoint(it->distance));
36             _sceneManager->getRootSceneNode()->addChild(nodeaux);
37           }
38       }
39       _selectedNode = it->movable->getParentSceneNode();
40       _selectedNode->showBoundingBox(true);
41     }
42   }
```

11.4.5. Máscaras

Las máscaras permiten restringir el ámbito de las consultas realizadas al gestor de escena. En el listado anterior, en las líneas 13 y 14 especificamos la máscara binaria que utilizaremos en la consulta (línea 6). Para que una máscara sea válida (y permita realizar operaciones lógicas con ella), debe contener un único uno.

Así, la forma más sencilla de definirlas es desplazando el 1 tantas posiciones como se indica tras el operador << como se indica en la Figura 11.38. Como el campo de máscara es de 32 bits, podemos definir 32 máscaras personalizadas para nuestra aplicación.

En el listado de la sección 11.4.4, vimos que mediante la llamada a `setQueryFlags` se podían asociar máscaras a entidades (líneas 5 y 19). Si no se especifica ninguna máscara en la creación de la entidad, ésta responderá a *todas* las *Queries*, planteando así un esquema bastante flexible.

Los operadores lógicos pueden emplearse para combinar varias máscaras, como el uso del AND `&&`, el OR `||` o la negación $\sim$ (ver líneas 13 y 14 del pasado código fuente). En este código fuente, la consulta a $\sim$`STAGE` sería equivalente a consultar por `CUBE1 | CUBE2`. La consulta de la línea 13 sería equivalente a preguntar por todos los objetos de la escena (no especificar ninguna máscara).

```
#define STAGE 1 << 0
#define CUBE1 1 << 1
#define CUBE2 1 << 2
```

Es equivalente a:

STAGE
0000000000000...00001
CUBE1
0000000000000...00010
CUBE2
0000000000000...00100

Figura 11.38: Máscaras binarias definidas en *MyFrameListener.h* para el ejemplo.

Capítulo 12

APIS de Gráficos 3D

Carlos González Morcillo

En este capítulo se introducirán los aspectos más relevantes de OpenGL. Como hemos señalado en el capítulo de introudcción, muchas bibliotecas de visualización (como OGRE) nos abstraen de los detalles de *bajo nivel*. Sin embargo, resulta interesante conocer el modelo de estados de este ipo de APIs, por compartir multitud de convenios con las bibliotecas de alto nivel. Entre otros aspectos, en este capítulo se estudiará la gestión de pilas de matrices y los diferentes modos de transformación de la API.

12.1. Introducción

Actualmente existen en el mercado dos alternativas principales como bibliotecas de representación 3D a bajo nivel: Direct3D y OpenGL. Estas dos bibliotecas son soportadas por la mayoría de los dispositivos hardware de aceleración gráfica. Direct3D forma parte del framework *DirectX* de *Microsoft*. Es una biblioteca ampliamente extendida, y multitud de motores 3D comerciales están basados en ella. La principal desventaja del uso de *DirectX* es la asociación exclusiva con el sistema operativo *Microsoft Windows*. Aunque existen formas de ejecutar un programa compilado para esta plataforma en otros sistemas, no es posible crear aplicaciones nativas utilizando *DirectX* en otros entornos. De este modo, en este primer estudio de las APIs de bajo nivel nos centraremos en la API multiplataforma OpenGL.

OpenGL es, probablemente, la biblioteca de programación gráfica más utilizada del mundo; desde videojuegos, simulación, CAD, visualización científica, y un largo etecétera de ámbitos de aplicación la configuran como la mejor alternativa en multitud de ocasiones. En esta sección se resumirán los aspectos básicos más relevantes para comenzar a utilizar OpenGL.

En este breve resumen de OpenGL se dejarán gran cantidad de aspectos sin mencionar. Se recomienda el estudio de la guía oficial de OpenGL [86] (también conocido como *El Libro Rojo de OpenGL* para profundizar en esta potente biblioteca gráfica. Otra fantástica fuente de información, con ediciones anteriores del *Libro Rojo* es la página oficial de la biblioteca[1].

El propio nombre *OpenGL* indica que es una *Biblioteca para Gráficos Abierta*[2]. Una de las características que ha hecho de OpenGL una biblioteca tan famosa es que es independiente de la plataforma sobre la que se está ejecutando (en términos software y hardware). Esto implica que *alguien* tendrá que encargarse de abrir una ventana gráfica sobre la que OpenGL pueda dibujar los preciosos gráficos 3D.

Para facilitar el desarrollo de aplicaciones con OpenGL sin preocuparse de los detalles específicos de cada sistema operativo (tales como la creación de ventanas, gestión de eventos, etc...) *M. Kilgard* creó GLUT, una biblioteca independiente de OpenGL (no forma parte de la distribución oficial de la API) que facilita el desarrollo de pequeños prototipos. Esta biblioteca auxiliar es igualmente multiplataforma. En este capítulo trabajaremos con FreeGLUT, la alternativa con licencia GPL totalmente compatible con GLUT. Si se desea mayor control sobre los eventos, y la posibilidad de extender las capacidades de la aplicación, se recomienda el estudio de otras APIs multiplataforma compatibles con OpenGL, como por ejemplo SDL[3] o la que emplearemos con OGRE (OIS). Existen alternativas específicas para sistemas de ventanas concretos (ver Figura 12.2, como *GLX* para plataformas Unix, *AGL* para Macintosh o *WGL* para sistemas Windows.

De este modo, el núcleo principal de las biblioteca se encuentra en el módulo *GL* (ver Figura 12.2). En el módulo GLU (OpenGL Utility) se encuentran funciones de uso común para el dibujo de diversos tipos de superficies (esferas, conos, cilindros, curvas...). Este módulo es parte oficial de la biblioteca.

Uno de los objetivos principales de OpenGL es la representación de imágenes de alta calidad a alta velocidad. OpenGL está diseñado para la realización de **aplicaciones interactivas**, como videojuegos. Gran cantidad de plataformas actuales se basan en esta especificación para definir sus interfaces de dibujado en gráficos 3D.

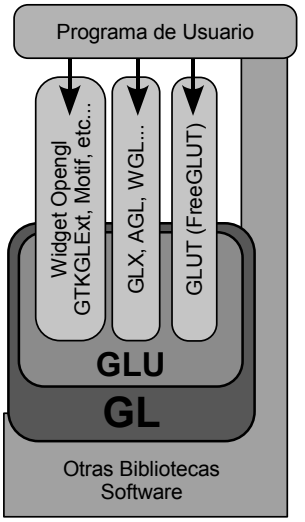


Figura 12.2: Relación entre los módulos principales de OpenGL.

12.2. Modelo Conceptual

Las llamadas a funciones de OpenGL están diseñadas para aceptar diversos tipos de datos como entrada. El nombre de la función identifica además los argumentos que recibirá. Por ejemplo, en la Figura 12.4 se llama a una función para especificar un nuevo vértice en coordenadas homogéneas (4 parámetros), con tipo de datos *double* y en formato vector. Se definen tipos enumerados como redefinición de tipos básicos (como *GLfloat*, *GLint*, etc) para facilitar la compatibilidad con otras plataformas. Es buena práctica utilizar estos tipos redefinidos si se planea compilar la aplicación en otros sistemas.

El **Modelo Conceptual General** de OpenGL define dos operaciones básicas que el programador puede realizar en cada instante; 1) dibujar algún elemento o 2) cambiar el estado de cómo se dibujan los elementos. La primera operación de *dibujar algún elemento* tiene que ser a) una primitiva geométrica (puntos, líneas o polígonos) o b) una primitiva de imagen.

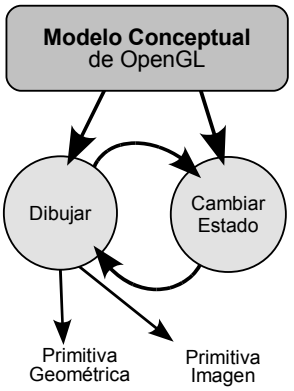


Figura 12.3: Modelo conceptual general de OpenGL.

[1] `http://www.opengl.org/`
[2] Las siglas de *GL* corresponden a *Graphics Library*
[3] `http://www.libsdl.org/`

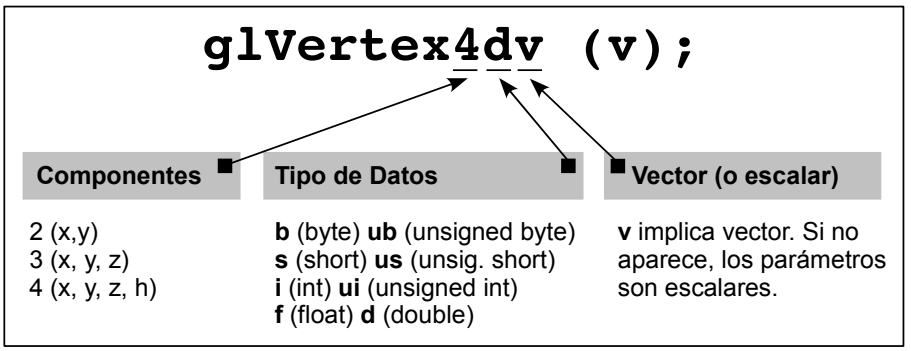


Figura 12.4: Prototipo general de llamada a función en OpenGL.

La Figura 12.3 resume el modelo conceptual de OpenGL. Así, la aplicación que utilice OpenGL será simplemente una colección de ciclos de cambio de estado y dibujado de elementos.

12.2.1. Cambio de Estado

Cambio de Estado

A modo de curiosidad: OpenGL cuenta con más de 400 llamadas a función que tienen que ver con el cambio del estado interno de la biblioteca.

La operación de **Cambiar el Estado** se encarga de inicializar las variables internas de OpenGL que definen cómo se dibujarán las primitivas. Este cambio de estado puede ser de mútiples tipos; desde cambiar el color de los vértices de la primitiva, establecer la posición de las luces, etc. Por ejemplo, cuando queremos dibujar el vértice de un polígono de color rojo, primero cambiamos el color del vértice con `glColor()` y después dibujamos la primitiva en ese nuevo estado con `glVertex()`.

Algunas de las formas más utilizadas para cambiar el estado de OpenGL son:

1. **Gestión de Vértices**: Algunas llamadas muy utilizadas cuando se trabaja con modelos poligonales son `glColor()` que utilizaremos en el primer ejemplo del capítulo para establecer el color con el que se dibujarán los vértices[4], `glNormal()` para especificar las normales que se utilizarán en la iluminación, o `glTexCoord()` para indicar coordendas de textura.

2. **Activación de Modos**: Mediante las llamadas a `glEnable` y `glDisable` se pueden activar o desactivar características internas de OpenGL. Por ejemplo, en la línea 3 del primer ejemplo del capítulo se activa el Test de Profundidad que utiliza y actualiza el Z-Buffer. Este test se utilizará hasta que de nuevo se cambia el *interruptor* desactivando esta funcionalidad en la línea 17.

3. **Características Especiales**: Existen multitud de características particulares de los elementos con los que se está trabajando. OpenGL define valores por defecto para los elementos con los que se está trabajando, que pueden ser cambiadas empleando llamadas a la API.

[4]Como veremos a continuación, los colores en OpenGL se especifican en punto flotante con valores entre 0 y 1. Las primeras tres componentes se corresponden con los canales RGB, y la cuarta es el valor de transparencia *Alpha*.

12.2.2. Dibujar Primitivas

La operación de **Dibujar Primitivas** requiere habitualmente que éstas se definan en coordenadas homogéneas. Todas las primitivas geométricas se especifican con vértices. El tipo de primitiva determina cómo se combinarán los vértices para formar la superfice poligional final. La creación de primitivas se realiza entre llamadas a `glBegin(PRIMITIVA)` y `glEnd()`, siendo `PRIMITIVA` alguna de las 10 primitivas básicas soportadas por OpenGL[5]. No obstante, el módulo GLU permite dibujar otras superficies más complejas (como cilindros, esferas, discos...), y con GLUT es posible dibujar algunos objetos simples. Como se puede ver, OpenGL no dispone de funciones para la carga de modelos poligonales creados con otras aplicaciones (como por ejemplo, en formato OBJ o MD3). Es responsabilidad del programador realizar esta carga y dibujarlos empleando las primitivas básicas anteriores.

Una vez estudiados los cambios de estado y el dibujado de primitivas básicas, podemos especificar en la función `display` del siguiente listado para que dibuje un cuadrado (primitiva `GL_QUADS`). Las líneas relativas al despliegue del cuadrado se corresponden con el intervalo `7-17`. Especificamos la posición de cada vértice del cuadrado modificando el color (el estado interno). OpenGL se encargará de calcular la transición de color entre cada punto intermedio del cuadrado.

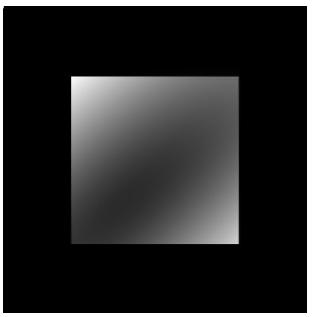


Figura 12.5: Salida por pantalla del ejemplo.

Listado 12.1: Ejemplo de cambio de estado y dibujo de primitivas.

```
1  void display() {
2      glClear( GL_COLOR_BUFFER_BIT );
3      glEnable(GL_DEPTH_TEST);
4      glLoadIdentity();         /* Cargamos la matriz identidad */
5      glTranslatef( 0.f, 0.f, -4.f );
6
7      glBegin(GL_QUADS);            /* Dibujamos un cuadrado    */
8      glColor3f(1.0, 0.0, 0.0);     /* de dos unidades de lado. */
9      glVertex3f(1.0, 1.0, 0.0);    /* Especificamos la coorde- */
10     glColor3f(0.0, 1.0, 0.0);     /* nada 3D de cada vertice  */
11     glVertex3f(1.0, -1.0, 0.0);   /* y su color asociado.     */
12     glColor3f(0.0, 0.0, 1.0);
13     glVertex3f(-1.0, -1.0, 0.0);
14     glColor3f(1.0, 1.0, 1.0);
15     glVertex3f(-1.0, 1.0, 0.0);
16     glEnd();
17     glDisable(GL_DEPTH_TEST);
18
19     glutSwapBuffers();
20 }
```

12.3. Pipeline de OpenGL

Como vimos en el capítulo 8.2, los elementos de la escena sufren diferentes transformaciones en el *pipeline* de gráficos 3D. Algunas de las principales etapas se corresponden con las siguiente operaciones:

- **Transformación de Modelado**: En la que los modelos se posicionan en la escena y se obtienen las Coordenadas Universales.

[5]Las 10 primitivas básicas de OpenGL son `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINES`, `GL_LINE_LOOP`, `GL_POLYGON`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLES`, `GL_TRIANGLE_FAN`, `GL_QUADS` y `GL_QUAD_STRIP`

- **Transformación de Visualización**: Donde se especifica la posición de la cámara y se mueven los objetos desde las coordenadas del mundo a las Coordenadas de Visualización (o coordenadas de cámara).
- **Transformación de Proyección**: Obteniendo Coordenadas Normalizadas en el cubo unitario.
- **Transformación de Recorte y de Pantalla**: Donde se obtienen, tras el recorte (o *clipping* de la geometría), las coordenadas 2D de la ventana en pantalla.

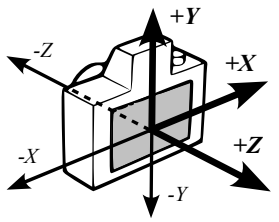


Figura 12.6: Descripción del sistema de coordenadas de la cámara definida en OpenGL.

OpenGL combina la Transformación de Modelado y la de Visualización en una Transformación llamada **"Modelview"**. De este modo OpenGL transforma directamente las Coordenadas Universales a Coordenadas de Visualización empleando la matriz *Modelview*. La posición inicial de la cámara en OpenGL sigue el convenio estudiado en el capítulo 9 y que se resume en la Figura 12.6.

12.3.1. Transformación de Visualización

La transformación de Visualización debe especificarse **antes** que ninguna otra transformación de Modelado. Esto es debido a que **OpenGL aplica las transformaciones en orden inverso**. De este modo, aplicando en el código las transformaciones de Visualización antes que las de Modelado nos aseguramos que ocurrirán después que las de Modelado.

Para comenzar con la definición de la Transformación de Visualización, es necesario *limpiar* la matriz de trabajo actual. OpenGL cuenta con una función que carga la matriz identidad como matriz actual `glLoadIdentity()`.

Matriz Identidad

La *Matriz Identidad* es una matriz 4x4 con valor 1 en la diagonal principal, y 0 en el resto de elementos. La multiplicación de esta matriz I por una matriz M cualquiera siempre obtiene como resultado M. Esto es necesario ya que OpenGL siempre multiplica las matrices que se le indican para modificar su estado interno.

Una vez hecho esto, podemos posicionar la cámara virtual de varias formas:

1. **glulookat**. La primera opción, y la más comunmente utilizada es mediante la función `gluLookAt`. Esta función recibe como parámetros un punto (*eye*) y dos vectores (*center* y *up*). El punto *eye* es el punto donde se encuentra la cámara en el espacio y mediante los vectores libres *center* y *up* orientamos hacia dónde mira la cámara (ver Figura 12.7).
2. **Traslación y Rotación**. Otra opción es especificar *manualmente* una secuencia de traslaciones y rotaciones para posicionar la cámara (mediante las funciones `glTraslate()` y `glRotate()`, que serán estudiadas más adelante.
3. **Carga de Matriz**. La última opción, que es la utilizada en aplicaciones de Realidad Aumentada, es la carga de la matriz de Visualización calculada *externamente*. Esta opción se emplea habitualmente cuando el programador quiere calcular la posición de la cámara virtual empleando métodos externos (como por ejemplo, mediante una biblioteca de *tracking* para aplicaciones de Realidad Aumentada).

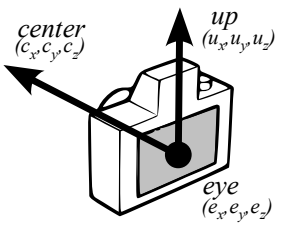


Figura 12.7: Parámetros de la función *glulookat*.

12.3.2. Transformación de Modelado

Las transformaciones de modelado nos permiten modificar los objetos de la escena. Existen tres operaciones básicas de modelado que implementa OpenGL con llamadas a funciones. No obstante, puede especificarse cualquier operación aplicando una matriz definida por el usuario.

- **Traslación**. El objeto se mueve a lo largo de un vector. Esta operación se realiza mediante la llamada a `glTranslate(x,y,z)`.

- **Rotación**. El objeto se rota en el eje definido por un vector. Esta operación se realiza mediante la llamada a `glRotate(`α`,x,y,z)`, siendo α el ángulo de rotación en grados sexagesimales (en sentido contrario a las agujas del reloj).
- **Escalado**. El objeto se escala un determinado valor en cada eje. Se realiza mediante `glScale(x,y,z)`.

12.3.3. Transformación de Proyección

Como hemos visto, las transformaciones de proyección definen el volumen de visualización y los planos de recorte. OpenGL soporta dos modelos básicos de proyección; la proyección ortográfica y la proyección en perspectiva.

El núcleo de OpenGL define una función para definir la pirámide de visualización (o *frustum*) mediante la llamada a `glFrustum()`. Esta función requiere los seis parámetros (t, b, r, l, f y n) estudiados en la sección 9.3.

Otro modo de especificar la transformación es mediante la función de GLU `gluPerspective(fov, aspect, near, far)`. En esta función, *far* y *near* son las distancias de los planos de recorte (igual que los parámetros f y n del *frustum*. *fov* especifica en grados sexagesimales el ángulo en el eje Y de la escena que es visible para el usuario, y *aspect* indica la relación de aspecto de la pantalla (ancho/alto).

12.3.4. Matrices

OpenGL utiliza matrices 4x4 para representar todas sus transformaciones geométricas. Las matrices emplean coordenadas homogéneas, como se estudió en la sección 9.2.1. A diferencia de la notación matemática estándar, OpenGL especifica por defecto las matrices por columnas, por lo que si queremos cargar nuestras propias matrices de transformación debemos tener en cuenta que el orden de elementos en OpenGL es el siguiente:

$$\begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix} \tag{12.1}$$

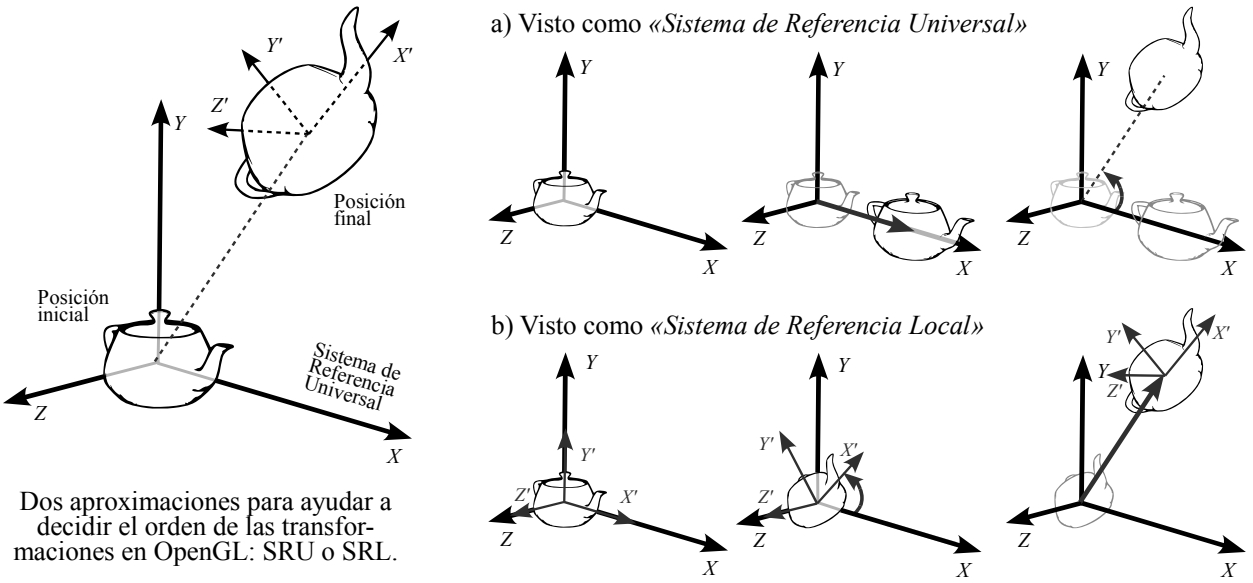
Sobre el uso de Matrices

Como el convenio en C y C++ de definición de matrices bidimensionales es ordenados por filas, suele ser una fuente de errores habitual definir la matriz como un array bidimensional. Así, para acceder al elemento superior derecho de la matriz, tendríamos que acceder al `matriz[3][0]` según la notación OpenGL. Para evitar errores, se recomienda definir el array como unidimensional de 16 elementos `GLfloat matriz[16]`.

OpenGL internamente maneja *pilas de matrices*, de forma que únicamente la matriz de la cima de cada pila es la que se está utilizando en un momento determinado. Hay cuatro pilas de matrices en OpenGL:

1. Pila *Modelview* (`GL_MODELVIEW`). Esta pila contiene las matrices de Transformación de modelado y visualización. Tiene un tamaño mínimo de 32 matrices (aunque, dependiendo del sistema puede haber más disponibles).
2. Pila *Projection* (`GL_PROJECTION`). Esta pila contiene las matrices de proyección. Tiene un tamaño mínimo de 2 elementos.
3. Pila *Color* (`GL_PROJECTION`). Utilizada para modificar los colores.
4. Pila *Texture* (`GL_TEXTURE`). Estas matrices se emplean para transformar las coordenadas de textura.

Es posible cambiar la pila sobre la que especificaremos las transformaciones empleando la llamada a `glMatrixMode()`. Por ejemplo, para utilizar la pila de *Modelview* utilizaríamos `glMatrixMode(GL_MODELVIEW)`.



Dos aproximaciones para ayudar a decidir el orden de las transformaciones en OpenGL: SRU o SRL.

Figura 12.8: Cómo decidir el orden de las operaciones en OpenGL.

El uso de *pilas* de matrices nos facilita la construcción de modelos jerárquicos, donde es posible utilizar modelos simples y ensamblarlos para construir modelos más complejos. Por ejemplo, si queremos dibujar un coche, tendríamos que dibujar las 4 ruedas con su posición relativa al chasis del vehículo. Así, dibujaríamos el chasis, y luego desplazándonos *cada vez desde el centro del coche* dibujaríamos cada una de las ruedas. Esta operación de *"volver al centro del coche"* se realiza fácilmente empleando pilas de matrices. Este concepto está directamente relacionado con el *Grafo de Escena* que estudiamos en el capítulo 10.

Recordemos que únicamente trabajamos con la matriz que está en la cima de la pila. La función `glPushMatrix()` añade una copia de la matriz de trabajo actual a la parte superior de la pila. Por su parte, la función `glPopMatrix()` elimina la matriz superior de la pila, descartando su información. La siguiente matriz de la pila pasará a ser la matriz *activa*. El efecto de esta función es el de "volver" al último punto que guardamos en la pila.

Supongamos ahora que queremos rotar un objeto 45º respecto del eje Z y trasladarlo 5 unidades respecto del eje X, obteniendo una determinada posición final (ver Figura 12.8 izquierda). El objeto inicialmente está en el origen del SRU. ¿En qué orden debemos aplicar las transformaciones de OpenGL? Como la transformación es de modelo, tendremos que aplicarla sobre la pila de matrices *Modelview*, con el siguiente código resultante:

Listado 12.2: Ejemplo de transformaciones.

```
1 glMatrixMode(GL_MODELVIEW);
2 glLoadIdentity();
3 glRotatef(45,0,0,1);
4 glTraslatef(5,0,0);
5 dibujar_objeto();
```

El código anterior dibuja el objeto en la posición deseada, pero ¿cómo hemos llegado a ese código y no hemos intercambiado las instrucciones de las líneas 3 y 4?. Existen dos formas de imaginarnos cómo se realiza el dibujado que nos puede ayudar a plantear el código fuente. Ambas formas son únicamente aproximaciones conceptuales, ya que el resultado en código debe ser exactamente el mismo.

- **Idea de Sistema de Referencia Universal Fijo**. La composición de movimientos aparece en orden inverso al que aparece en el código fuente. Esta idea es como ocurre realmente en OpenGL. Las transformaciones se aplican siempre respecto del SRU, y en orden inverso a como se indica en el código fuente. De este modo, la primera transformación que ocurrirá será la traslación (línea 4) y después la rotación *respecto del origen del sistema de referencia universal* (línea 3). El resultado puede verse en la secuencia de la Figura 12.8 a).
- **Idea del Sistema de Referencia Local**. También podemos imaginar que cada objeto tiene un sistema de referencia local *interno* al objeto que va cambiando. La composición se realiza en el mismo orden que aparece en el código fuente, y siempre respecto de ese sistema de referencia local. De esta forma, como se muestra en la secuencia de la Figura 12.8 b), el objeto primero rota (línea 3) por lo que su sistema de referencia local queda rotado respecto del SRU, y respecto de ese sistema de referencia local, posteriormente lo trasladamos 5 unidades respecto del eje X' (local).

Como hemos visto, es posible además cargar matrices de transformación definidas por nuestros propios métodos (podría ser interesante si empleáramos, por ejemplo, algún método para calcular la proyección de sombras). Esto se realiza con la llamada a función `glLoadMatrix()`. Cuando se llama a esta función se reemplaza la cima de la pila de matrices activa con el contenido de la matriz que se pasa como argumento.

Si nos interesa es multiplicar una matriz definida en nuestros métodos por el contenido de la cima de la pila de matrices, podemos utilizar la función `glMultMatrix` que postmultiplica la matriz que pasamos como argumento por la matriz de la cima de la pila.

12.3.5. Dos ejemplos de transformaciones jerárquicas

Ejemplo Planetario

Ejemplo sencillo para afianzar el orden de las transformaciones.

Veamos a continuación un ejemplo sencillo que utiliza transformaciones jerárquicas. Definiremos un sistema planetario que inicialmente estará formado por el Sol y la Tierra.

Listado 12.3: Sistema Planetario

```
1  // ==== Definicion de constantes y variables globales ==============
2  long hours = 0;   // Horas transcurridas (para calculo rotaciones)
3  // ======== display ================================================
4  void display()
5  {
6    float RotEarthDay=0.0;  // Movimiento de rotacion de la tierra
7    float RotEarth=0.0;     // Movimiento de traslacion de la tierra
8    glClear( GL_COLOR_BUFFER_BIT );
9    glPushMatrix();
10
11   RotEarthDay = (hours % 24) * (360/24.0);
12   RotEarth = (hours / 24.0) * (360 / 365.0) * 10; // x10 rapido!
13
14   glColor3ub (255, 186, 0);
15   glutWireSphere (1, 16, 16);             // Sol (radio 1 y 16 div)
16   glRotatef (RotEarth, 0.0, 0.0, 1.0);
17   glTranslatef(3, 0.0, 0.0);              // Distancia Sol, Tierra
```

```
18   glRotatef (RotEarthDay, 0.0, 0.0, 1.0);
19   glColor3ub (0, 0, 255);
20   glutWireSphere (0.5, 8, 8);             // Tierra (radio 0.5)
21   glutSwapBuffers();
22   glPopMatrix();
23 }
```

Como puede verse en el listado anterior, se ha incluido una variable global `hours` que se incrementa cada vez que se llama a la función `display`. En este ejemplo, esta función se llama cada vez que se pulsa cualquier tecla. Esa variable modela el paso de las horas, de forma que la traslación y rotación de la *Tierra* se calculará a partir del número de horas que han pasado (líneas 11 y 12). En la simulación se ha acelerado 10 veces el movimiento de traslación para que se vea más claramente.

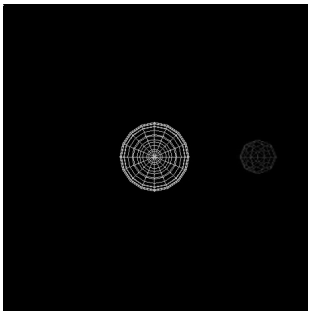


Figura 12.9: Salida por pantalla del ejemplo del planetario.

Empleando la *Idea de Sistema de Referencia Local* podemos pensar que desplazamos el sistema de referencia para dibujar el sol. En la línea 15, para dibujar una esfera alámbrica especificamos como primer argumento el radio, y a continuación el número de *rebanadas* (horizontales y verticales) en que se dibujará la esfera.

En ese punto dibujamos la primera esfera correspondiente al Sol. Hecho esto, rotamos los grados correspondientes al movimiento de traslación de la tierra (línea 16) y nos desplazamos 3 unidades respecto del eje X local del objeto (línea 17). Antes de dibujar la *Tierra* tendremos que realizar el movimiento de rotación de la tierra (línea 18). Finalmente dibujamos la esfera en la línea 20.

Veamos ahora un segundo ejemplo que utiliza `glPushMatrix()` y `glPopMatrix()` para dibujar un brazo robótico sencillo. El ejemplo simplemente emplea dos cubos (convenientemente escalados) para dibujar la estructura jerárquica de la figura 12.10.

Ejemplo Brazo Robot

Un segundo ejemplo que utiliza las funciones de añadir y quitar elementos de la pila de matrices.

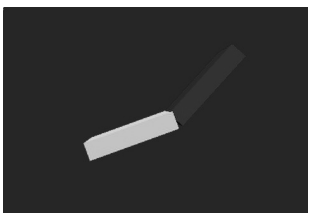


Figura 12.10: Salida por pantalla del ejemplo del robot.

En este ejemplo, se asocian manejadores de *callback* para los eventos de teclado, que se corresponden con la función `keyboard` (en 8-16). Esta función simplemente incrementa o decrementa los ángulos de rotación de las dos articulaciones del robot, que están definidas como variables globales en 2 y 3.

La parte más *interesante* del ejemplo se encuentra definido en la función `dibujar` en las líneas 19-42.

Listado 12.4: Brazo Robótico

```
1  // ==== Definicion de constantes y variables globales ==============
2  static int hombro = 15;
3  static int codo = 30;
4  GLfloat matVerde[] = {0.0, 1.0, 0.0, 1.0};
5  GLfloat matAzul[] = {0.0, 0.0, 1.0, 1.0};
6
7  // ==== Funcion de callback del teclado ============================
8  void teclado(unsigned char key, int x, int y) {
9    switch (key) {
10   case 'q': hombro = (hombro++) % 360; break;
11   case 'w': hombro = (hombro--) % 360; break;
12   case 'a': codo = (codo++) % 360; break;
13   case 's': codo = (codo--) % 360; break;
14   }
15   glutPostRedisplay();
16 }
17
18 // ==== Funcion de dibujado ========================================
19 void dibujar() {
20   glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
21   glLoadIdentity();
22   gluLookAt(0.0, 1.0, 6.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0);
23
24   glRotatef(hombro, 0.0, 0.0, 1.0);
25   glTranslatef(1.0, 0.0, 0.0);  // Nos posicionamos en la mitad
```

```
26    glPushMatrix();               // Guardamos la posicion
27    glScalef (2.0, 0.7, 0.1);     // Establecemos la escala
28    glMaterialfv(GL_FRONT, GL_DIFFUSE, matVerde);
29    glutSolidCube(1.0);           // Dibujamos el çubo"
30    glPopMatrix();                // Recuperamos la posicion
31
32    glTranslatef(1.0,0.0,0.0);    // Continuamos hasta el extremo
33    glRotatef(codo, 0.0, 0.0, 1.0);
34    glTranslatef(1.0,0.0,0.0);    // Nos posicionamos en la mitad
35    glPushMatrix();               // Guardamos la posicion
36    glScalef (2.0, 0.7, 0.1);     // Establecemos la .escala"
37    glMaterialfv(GL_FRONT, GL_DIFFUSE, matAzul);
38    glutSolidCube(1.0);           // Dibujamos el çubo"
39    glPopMatrix();
40
41    glutSwapBuffers();
42  }
```

Aplicamos de nuevo la idea de trabajar en un sistema de referencia local que se desplaza con los objetos según los vamos dibujando. De este modo nos posicionaremos en la mitad del trayecto para dibujar el cubo *escalado*. El cubo siempre se dibuja en el centro del sistema de referencia local (dejando mitad y mitad del cubo en el lado positivo y negativo de cada eje). Por tanto, para que el cubo *rote* respecto del extremo tenemos que rotar primero (como en la línea 24) y luego desplazarnos hasta la mitad del trayecto (línea 25), dibujar y recorrer la otra mitad 32 antes de dibujar la segunda parte del robot.

12.4. Ejercicios Propuestos

Se recomienda la realización de los ejercicios de esta sección en orden, ya que están relacionados y su complejidad es ascendente.

1. Modifique el ejemplo del listado del planetario para que dibuje una esfera de color blanco que representará a la *Luna*, de radio 0.2 y separada 1 unidad del centro de la *Tierra* (ver Figura 12.11). Este objeto tendrá una rotación completa alrededor de la *Tierra* cada 2.7 días (10 veces más rápido que la rotación real de la *Luna* sobre la *Tierra*. Supondremos que la luna no cuenta con rotación interna[6].

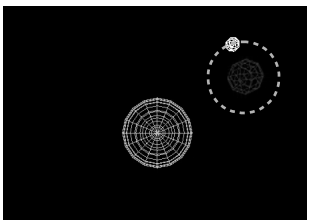


Figura 12.11: Ejemplo de salida del ejercicio propuesto.

2. Modifique el ejemplo del listado del brazo robótico para que una base (añadida con un toroide `glutSolidTorus`) permita rotar el brazo robótico respecto de la base (eje Y) mediante las teclas *1* y *2*. Además, añada el código para que el extremo cuente con unas pinzas (creadas con `glutSolidCone`) que se abran y cierren empleando las teclas *z* y *x*, tal y como se muestra en la Figura 12.12.

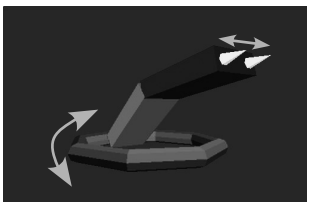


Figura 12.12: Ejemplo de salida del ejercicio del robot.

[6]Nota: Para la resolución de este ejercicio es recomendable utilizar las funciones de `glPushMatrix()` y `glPopMatrix()` para volver al punto donde se dibujó la *Tierra* antes de aplicar su rotación interna.

Capítulo 13

Gestión Manual OGRE 3D

Carlos González Morcillo

En las sesiones anteriores hemos delegado la gestión del arranque, incialización y parada de Ogre en una clase *SimpleExample* proporcionada junto con el motor gráfico para facilitar el desarrollo de los primeros ejemplos. En este capítulo estudiaremos la gestión semi-automática de la funcionalidad empleada en los ejemplos anteriores, así como introduciremos nuevas características, como la creación manual de entidades, y el uso de *Overlays*, luces y sombras dinámicas.

13.1. Inicialización Manual

Inicio Casi-Manual

En este capítulo no describiremos el inicio totalmente manual de Ogre; emplearemos las llamadas de alto nivel para cargar plugins, inicializar ventana, etc...

En esta sección introduciremos un esqueleto de código fuente que emplearemos en el resto del capítulo, modificándolo de manera incremental. Trabajaremos con dos clases; `MyApp` que proporciona el núcleo principal de la aplicación, y la clase `MyFrameListener` que es una isntancia de clase que hereda de `Ogre::FrameListener`, basada en el patrón *Observador*.

Como puede verse en el primer listado, en el fichero de cabecera de `MyApp.h` se declara la clase que contiene tres miembros privados; la instancia de *root* (que hemos estudiado en capítulos anteriores), el gestor de escenas y un puntero a un objeto de la clase propia *MySceneManager* que definiremos a continuación.

Listado 13.1: MyApp.h

```
1 #include <Ogre.h>
2 #include "MyFrameListener.h"
3
4 class MyApp {
5 private:
```

```
 6   Ogre::SceneManager* _sceneManager;
 7   Ogre::Root* _root;
 8   MyFrameListener* _framelistener;
 9
10 public:
11   MyApp();
12   ~MyApp();
13   int start();
14   void loadResources();
15   void createScene();
16 };
```

El programa principal únicamente tendrá que crear una instancia de la clase `MyApp` y ejecutar su método `start`. La definición de este método puede verse en el siguiente listado, en las líneas 13-44.

13.1.1. Inicialización

En la línea 14 se crea el objeto *Root*. Esta es la primera operación que se debe realizar antes de ejecutar cualquier otra operación con Ogre. El constructor de *Root* está sobrecargado, y permite que se le pase como parámetros el nombre del archivo de configuración de plugins (por defecto, buscará *plugins.cfg* en el mismo directorio donde se encuentra el ejecutable), el de configuración de vídeo (por defecto *ogre.cfg*), y de log (por defecto *ogre.log*). Si no le indicamos ningún parámetro (como en el caso de la línea 14), tratará de cargar los ficheros con el nombre por defecto.

El comportamiento del constructor de *Root* es diferente entre no especificar parámetro, o pasar la cadena vacía `""`. En el primer caso se buscarán los archivos con el nombre por defecto. En el segundo caso, indicamos a Ogre que cargaremos *manualmente* los plugins y la configuración de vídeo.

En la línea 16 se intenta cargar la configuración de vídeo existente en el archivo *ogre.cfg*. Si el archivo no existe, o no es correcto, podemos abrir el diálogo que empleamos en los ejemplos de las sesiones anteriores (línea 17), y guardar a continuacion los valores elegidos por el usuario (línea 18). Con el objeto *Root* creado, podemos pasar a crear la ventana. Para ello, en la línea 21 solicitamos al objeto *Root* que finalice su inicialización creando una ventana que utilice la configuración que eligió el usuario, con el título especificado como segundo parámetro (*MyApp* en este caso).

El primer parámetro booleano del método indica a Ogre que se encarge de crear automáticamente la ventana. La llamada a este método nos devuelve un puntero a un objeto *RenderWindow*, que guardaremos en la variable `window`.

Lo que será representado en la ventana vendrá definido por el volumen de visualización de la cámara virtual. Asociado a esta cámara, crearemos una *superficie* (algo que conceptualmente puede verse como el lienzo, o el *plano de imagen*) sobre la que se dibujarán los objetos 3D. Esta superficie se denomina *viewport*. El *Gestor de Escena* admite diversos modos de gestión, empleando el patrón de *Factoría*, que son cargados como plugins (y especificados en el archivo *plugins.cfg*). El parámetro indicado en la

llamada de (22) especifica el tipo de gestor de escena que utilizaremos ST_GENERIC[1]. Esta gestión de escena mínima no está optimizada para ningún tipo de aplicación en particular, y resulta especialmente adecuada para pequeñas aplicaciones, menús de introducción, etc.

Listado 13.2: MyApp.cpp

```
1  #include "MyApp.h"
2
3  MyApp::MyApp() {
4    _sceneManager = NULL;
5    _framelistener = NULL;
6  }
7
8  MyApp::~MyApp() {
9    delete _root;
10   delete _framelistener;
11 }
12
13 int MyApp::start() {
14   _root = new Ogre::Root();       // Creamos el objeto root
15
16   if(!_root->restoreConfig()) {   // Si no se puede restaurar
17     _root->showConfigDialog();    // Abrimos ventana de config
18     _root->saveConfig();          // Guardamos la configuracion
19   }
20
21   Ogre::RenderWindow* window = _root->initialise(true,"MyApp");
22   _sceneManager = _root->createSceneManager(Ogre::ST_GENERIC);
23
24   Ogre::Camera* cam = _sceneManager->createCamera("MainCamera");
25   cam->setPosition(Ogre::Vector3(5,20,20));
26   cam->lookAt(Ogre::Vector3(0,0,0));
27   cam->setNearClipDistance(5);    // Establecemos distancia de
28   cam->setFarClipDistance(10000); // planos de recorte
29
30   Ogre::Viewport* viewport = window->addViewport(cam);
31   viewport->setBackgroundColour(Ogre::ColourValue(0.0,0.0,0.0));
32   double width = viewport->getActualWidth();
33   double height = viewport->getActualHeight();
34   cam->setAspectRatio(width / height);
35
36   loadResources();   // Metodo propio de carga de recursos
37   createScene();     // Metodo propio de creacion de la escena
38
39   _framelistener = new MyFrameListener();   // Clase propia
40   _root->addFrameListener(_framelistener);  // Lo anadimos!
41
42   _root->startRendering();    // Gestion del bucle principal
43   return 0;                   // delegada en OGRE
44 }
45
46 void MyApp::loadResources() {
47   Ogre::ConfigFile cf;
48   cf.load("resources.cfg");
49
50   Ogre::ConfigFile::SectionIterator sI = cf.getSectionIterator();
51   Ogre::String sectionstr, typestr, datastr;
52   while (sI.hasMoreElements()) {    // Mientras tenga elementos...
53     sectionstr = sI.peekNextKey();
54     Ogre::ConfigFile::SettingsMultiMap *settings = sI.getNext();
55     Ogre::ConfigFile::SettingsMultiMap::iterator i;
56     for (i = settings->begin(); i != settings->end(); ++i) {
57       typestr = i->first;    datastr = i->second;
58       Ogre::ResourceGroupManager::getSingleton().
             addResourceLocation(datastr, typestr, sectionstr);
```

[1]Existen otros métodos de gestión, como ST_INTERIOR, ST_EXTERIOR... Estudiaremos en detalle las opciones de estos gestores de escena a lo largo de este módulo.

```
59      }
60    }
61    Ogre::ResourceGroupManager::getSingleton().
          initialiseAllResourceGroups();
62  }
63
64  void MyApp::createScene() {
65    Ogre::Entity* ent = _sceneManager->createEntity("Sinbad.mesh");
66    _sceneManager->getRootSceneNode()->attachObject(ent);
67  }
```

El gestor de escena es el encargado de crear las cámaras virtuales. En realidad el gestor de escena trabaja como una factoría de diferentes tipos de objetos que crearemos en la escena. En la línea [24] obtenemos un puntero a una cámara que llamaremos *MainCamera*. A continuación establecemos la posición de la cámara virtual (línea [25]), y el punto hacia el que mira del SRU (línea [26]). En las líneas [27-28] establecemos la distancia con los planos de recorte cercano y lejano (ver Sección 9.3). En la línea [34] se establece la relación de aspecto de la cámara. Esta relación de aspecto se calcula como el número de píxeles en horizontal con respecto del número de píxeles en vertical. Resoluciones de 4/3 (como 800x600, o 1024x768) tendrán asociado un valor de ratio de 1.3333, mientras que resoluciones panorámicas de 16/9 tendrán un valor de 1.77, etc.

Para calcular el aspect ratio de la cámara, creamos un *viewport* asociado a esa cámara y establecemos el color de fondo como negro (líneas [30-31]).

Viewports...

Una cámara puede tener cero o más *Viewports*. Un uso común de los viewports es la generación de imágenes dentro de otros viewports. Veremos un uso de esta característica en próximos capítulos del módulo.

A continuación en las líneas [36-37] se ejecutan métodos propios de la clase para cargar los recursos y crear la escena. La carga de recursos manual la estudiaremos en la sección 13.1.2. La creación de la escena (líneas [64-67]), se realiza añadiendo la entidad directamente al nodo *Root*.

Las líneas [39-40] crean un objeto de tipo *FrameListener*. Una clase *FrameListener* es cualquier clase que implemente el interfaz de *FrameListener*, permitiendo que Ogre llame a ciertos métodos al inicio y al final del dibujado de cada frame empleando el patrón *Observer*. Veremos en detalle cómo se puede implementar esta clase en la sección 13.1.3.

Al finalizar la inicialización, llamamos al método *startRendering* de *Root* (línea [42]), que inicia el bucle principal de rendering. Esto hace que Ogre entre en un bucle infinito, ejecutando los métodos asociados a los *FrameListener* que hayan sido añadidos anteriormente.

En muchas ocasiones, no es conveniente delegar en Ogre la gestión del bucle principal de dibujado. Por ejemplo, si queremos atender peticiones de networking, o actualizar estados internos del juego. Además, si queremos integrar la ventana de visualización de Ogre como un widget dentro de un entorno de ventanas, es posible que éste no nos deje utilizar nuestro propio bucle pricipal de dibujado. En este caso, como veremos en sucesivos ejemplos, puede ser conveniente emplear la llamada a `renderOneFrame()` y gestionar manualmente el bucle de rendering.

13.1.2. Carga de Recursos

El fichero de definición de recursos se especifica en la línea 48. Este archivo está formado de diferentes secciones, que contienen pares de clave y valor. En el sencillo ejemplo que vamos a definir en este capítulo, tendremos el contenido definido en la Figura 13.1.

```
[General]
FileSystem=media
```

Figura 13.1: Contenido del archivo `resources.cfg` del ejemplo.

Como puede verse, la única sección del archivo se denomina *General*. Crearemos un iterador que permita cargar cada una de las secciones y, dentro de cada una, obtendremos los elementos que la definen.

La línea 50 crea el *SectionIterator* asociado al fichero de configuración. Mientras existan elementos que procesar por el iterador (bucle *while* de las líneas 52-60), obtiene la **clave** del primer elemento de la colección (línea 53) sin avanzar al siguiente (en el caso del ejemplo, obtiene la sección *General*). A continuación, obtiene en *settings* un puntero al **valor** del elemento actual de la colección, *avanzando al siguiente*. Este valor en realidad otro mapa. Emplearemos un segundo iterador (líneas 56-69) para recuperar los nombres y valores de cada entrada de la sección. En el fichero de ejemplo, únicamente iteraremos una vez dentro del segundo iterador, para obtener en `typestr` la entrada a `FileSystem`, y en `datastr` el valor del directorio `media`.

Finalmente la llamada al *ResourceGroupManager* (línea 61) solicita que se inicialicen todos los grupos de recursos que han sido añadidos en el iterador anterior, en la línea 58.

13.1.3. FrameListener

Como hemos comentado anteriormente, el uso de *FrameListener* se basa en el patrón *Observador*. Añadimos instancias de esta clase al método Root de forma que será notificada cuando ocurran ciertos eventos. Antes de representar un frame, Ogre itera sobre todos los *FrameListener* añadidos, ejecutando el método *frameStarted* de cada uno de ellos. Cuando el frame ha sido dibujado, Ogre ejecutará igualmente el método *frameEnded* asociado a cada *FrameListener*. En el siguiente listado se declara la clase `MyFrameListener`.

FrameStarted

Si delegamos la gestión de bucle de dibujado a Ogre, la gestión de los eventos de teclado, ratón y joystick se realiza en *FrameStarted*, de modo que se atienden las peticiones *antes* del dibujado del frame.

Listado 13.3: MyFrameListener.h

```
1 #include <OgreFrameListener.h>
2
3 class MyFrameListener : public Ogre::FrameListener {
4 public:
5   bool frameStarted(const Ogre::FrameEvent& evt);
6   bool frameEnded(const Ogre::FrameEvent& evt);
7 };
```

Vemos que en esta sencilla implementación de la clase *MyFrameListener* no es necesario tener ninguna variable miembro específica, ni definir ningún constructor o destructor particular asociado a la clase. Veremos en las siguientes secciones cómo se complica la implementación de la misma según necesitamos añadir nueva funcionalidad.

La definición de la clase es igualmente sencilla. Los métodos *frameStarted* o *frameEnded* devolverán *false* cuando queramos finalizar la aplicación. En este ejemplo, tras llamar la primera vez a *frameStarted*, se enviará al terminal la cadena `Frame Started` y Ogre finalizará. De este modo, no se llegará a imprimir la cadena asociada

a *frameEnded* (ni se representará el primer frame de la escena!). Si modificamos la implementación del método *frameStarted*, devolviendo *true*, se dibujará el primer frame y al ejecutar el método *frameEnded* (devolviendo *false*), Ogre finalizará el bucle principal de dibujado.

Habitualmente el método *frameEnded* se define pocas veces. Únicamente si necesitamos liberar memoria, o actualizar algún valor tras dibujar el frame, se realizará una implementación específica de este método. En los próximos ejemplos, únicamente definiremos el método *frameStarted*.

Listado 13.4: MyFrameListener.cpp

```
1  #include "MyFrameListener.h"
2
3  bool MyFrameListener::frameStarted(const Ogre::FrameEvent& evt) {
4    std::cout << "Frame started" << std::endl;
5    return false;
6  }
7
8  bool MyFrameListener::frameEnded(const Ogre::FrameEvent& evt)  {
9    std::cout << "Frame ended" << std::endl;
10   return false;
11 }
```

13.2. Uso de OIS

A continuación utilizaremos OIS para añadir un soporte de teclado mínimo, de forma que cuando se pulse la tecla ESC, se cierre la aplicación. Bastará con devolver *false* en el método *frameStarted* del *FrameListener* cuando esto ocurra. Como vemos en el siguiente listado, es necesario añadir al constructor de la clase (línea 11) un puntero a la *RenderWindow*.

Sobre OIS

Recordemos que OIS no forma parte de la distribución de Ogre. Es posible utilizar cualquier biblioteca para la gestión de eventos.

Listado 13.5: MyFrameListener.h

```
1  #include <OgreFrameListener.h>
2  #include <OgreRenderWindow.h>
3  #include <OIS/OIS.h>
4
5  class MyFrameListener : public Ogre::FrameListener {
6  private:
7    OIS::InputManager* _inputManager;
8    OIS::Keyboard* _keyboard;
9
10 public:
11   MyFrameListener(Ogre::RenderWindow* win);
12   ~MyFrameListener();
13   bool frameStarted(const Ogre::FrameEvent& evt);
14 };
```

Esta ventana se necesita para obtener el *manejador* de la ventana. Este manejador es un identificador único que mantiene el sistema operativo para cada ventana. Cada sistema operativo mantiene una lista de atributos diferente. Ogre abstrae del sistema operativo subyacente empleando una función genérica, que admite como primer parámetro el tipo de elemento que queremos obtener, y como segundo un puntero al tipo de datos que vamos a recibir. En el caso de la llamada de la línea 7, obtenemos el manejador de la ventana (el tipo asociado al manejador se especifica mediante la cadena `WINDOW`).

En las líneas 8-9 convertimos el manejador a cadena, y añadimos el par de objetos (empleando la plantilla *pair* de std) como parámetros de OIS. Como puede verse el convenio de Ogre y de OIS es similar a la hora de nombrar el manejador de la ventana (salvo por el hecho de que OIS requiere que se especifique como cadena, y Ogre lo devuelve como un entero).

Con este parámetro, creamos un *InputManager* de OIS en 11, que nos permitirá crear diferentes interfaces para trabajar con eventos de teclado, ratón, etc. Así, en las líneas 12-13 creamos un objeto de tipo *OIS::Keyboard*, que nos permitirá obtener las pulsaciones de tecla del usuario. El segundo parámetro de la línea 13 permite indicarle a OIS si queremos que almacene en un buffer los eventos de ese objeto. En este caso, la entrada por teclado se consume directamente sin almacenarla en un buffer intermedio.

Tanto el *InputManager* como el ojbeto de teclado deben ser eliminados explícitamente en el destructor de nuestro *FrameListener*. Ogre se encarga de liberar los recursos asociados a todos sus *Gestores*. Como Ogre es independiente de OIS, no tiene constancia de su uso y es responsabilidad del programador liberar los objetos de entrada de eventos, así como el propio gestor *InputManager*.

Listado 13.6: MyFrameListener.cpp

```
1  #include "MyFrameListener.h"
2
3  MyFrameListener::MyFrameListener(Ogre::RenderWindow* win) {
4    OIS::ParamList param;
5    unsigned int windowHandle;  std::ostringstream wHandleStr;
6
7    win->getCustomAttribute("WINDOW", &windowHandle);
8    wHandleStr << windowHandle;
9    param.insert(std::make_pair("WINDOW", wHandleStr.str()));
10
11   _inputManager = OIS::InputManager::createInputSystem(param);
12   _keyboard = static_cast<OIS::Keyboard*>
13     (_inputManager->createInputObject(OIS::OISKeyboard, false));
14 }
15
16 MyFrameListener::~MyFrameListener() {
17   _inputManager->destroyInputObject(_keyboard);
18   OIS::InputManager::destroyInputSystem(_inputManager);
19 }
20
21 bool MyFrameListener::frameStarted(const Ogre::FrameEvent& evt) {
22   _keyboard->capture();
23   if(_keyboard->isKeyDown(OIS::KC_ESCAPE)) return false;
24   return true;
25 }
```

La implementación del método *frameStarted* es muy sencilla. En la línea 22 se obtiene si hubo alguna pulsación de tecla. Si se presionó la tecla Escape (línea 23), se devuelve *false* de modo que Ogre finalizará el bucle principal de dibujado y liberará todos los recursos que se están empleando.

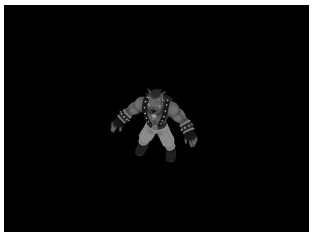


Figura 13.2: Resultado de ejecución del ejemplo básico con OIS.

Hasta ahora, la escena es totalmente estática. La Figura 13.2 muestra el resultado de ejecutar el ejemplo (hasta que el usuario presiona la tecla ESC). A continuación veremos cómo modificar la posición de los elementos de la escena, definiendo un interfaz para el manejo del ratón.

13.2.1. Uso de Teclado y Ratón

En el ejemplo de esta sección desplazaremos la cámara y rotaremos el modelo empleando el teclado y el ratón. Como en el diseño actual de nuestra apliación el despliegue se gestiona íntegramente en la clase *MyFrameListener*, será necesario que esta clase *conozca* los nuevos objetos sobre los que va a trabajar: el ratón (línea 8), la cámara (declarada igualmente como variable miembro privada en 9), y el nodo (que contendrá las entidades que queremos mover en 10).

En estos primeros ejemplos nos centraremos en el aspecto funcional de los mismos. En sucesivos capítulos estudiaremos otras aproximaciones de diseño para la gestión de eventos.

Listado 13.7: MyFrameListener.h

```
1  #include <Ogre.h>
2  #include <OIS/OIS.h>
3
4  class MyFrameListener : public Ogre::FrameListener {
5  private:
6    OIS::InputManager* _inputManager;
7    OIS::Keyboard* _keyboard;
8    OIS::Mouse* _mouse;
9    Ogre::Camera* _camera;
10   Ogre::SceneNode *_node;
11
12 public:
13   MyFrameListener(Ogre::RenderWindow* win, Ogre::Camera* cam,
14           Ogre::SceneNode* node);
15   ~MyFrameListener();
16   bool frameStarted(const Ogre::FrameEvent& evt);
17 };
```

Listado 13.8: MyFrameListener.cpp

```
1  #include "MyFrameListener.h"
2
3  MyFrameListener::MyFrameListener(Ogre::RenderWindow* win,
4       Ogre::Camera* cam,  Ogre::SceneNode *node) {
5    OIS::ParamList param;
6    size_t windowHandle;  std::ostringstream wHandleStr;
7
8    _camera = cam;  _node = node;
9
10   win->getCustomAttribute("WINDOW", &windowHandle);
11   wHandleStr << windowHandle;
12   param.insert(std::make_pair("WINDOW", wHandleStr.str()));
13
14   _inputManager = OIS::InputManager::createInputSystem(param);
15   _keyboard = static_cast<OIS::Keyboard*>
16     (_inputManager->createInputObject(OIS::OISKeyboard, false));
17   _mouse = static_cast<OIS::Mouse*>
18     (_inputManager->createInputObject(OIS::OISMouse, false));
19 }
20
21 MyFrameListener::~MyFrameListener() {
22   _inputManager->destroyInputObject(_keyboard);
23   _inputManager->destroyInputObject(_mouse);
24   OIS::InputManager::destroyInputSystem(_inputManager);
25 }
26
27 bool MyFrameListener::frameStarted(const Ogre::FrameEvent& evt) {
28   Ogre::Vector3 vt(0,0,0);     Ogre::Real tSpeed = 20.0;
29   Ogre::Real r = 0;
30   Ogre::Real deltaT = evt.timeSinceLastFrame;
31
32   _keyboard->capture();
```

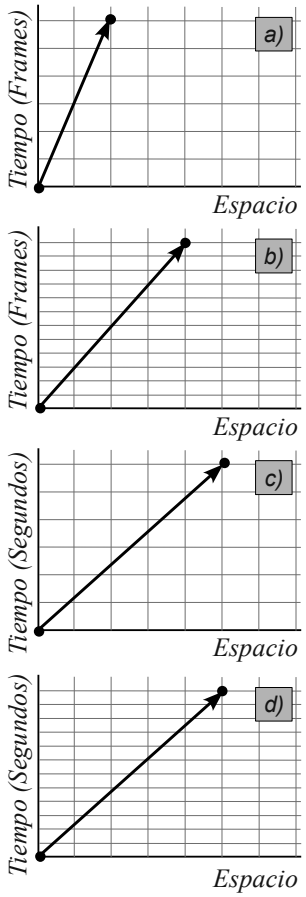


Figura 13.3: Comparación entre animación basada en frames y animación basada en tiempo. El eje de abcisas representa el espacio recorrido en una trayectoria, mientras que el eje de ordenadas representa el tiempo. En las gráficas a) y b) el tiempo se especifica en frames. En las gráficas c) y d) el tiempo se especifica en segundos. Las gráficas a) y c) se corresponden con los rsultados obtenidos en un computador con bajo rendimiento gráfico (en el mismo tiempo presenta una baja tasa de frames por segundo). Las gráficas b) y d) se corresponden a un computador con el doble de frames por segundo.

```
33   if(_keyboard->isKeyDown(OIS::KC_ESCAPE)) return false;
34   if(_keyboard->isKeyDown(OIS::KC_UP))   vt+=Ogre::Vector3(0,0,-1);
35   if(_keyboard->isKeyDown(OIS::KC_DOWN)) vt+=Ogre::Vector3(0,0,1);
36   if(_keyboard->isKeyDown(OIS::KC_LEFT)) vt+=Ogre::Vector3(-1,0,0);
37   if(_keyboard->isKeyDown(OIS::KC_RIGHT))vt+=Ogre::Vector3(1,0,0);
38   _camera->moveRelative(vt * deltaT * tSpeed);
39
40   if(_keyboard->isKeyDown(OIS::KC_R)) r+=180;
41   _node->yaw(Ogre::Degree(r * deltaT));
42
43   _mouse->capture();
44   float rotx = _mouse->getMouseState().X.rel * deltaT * -1;
45   float roty = _mouse->getMouseState().Y.rel * deltaT * -1;
46   _camera->yaw(Ogre::Radian(rotx));
47   _camera->pitch(Ogre::Radian(roty));
48
49   return true;
50 }
```

La implementación de la clase *MyFrameListener* requiere asignar en el constructor los punteros de la cámara y del nodo de escena a las variables miembro privadas (línea 8). De forma análoga, crearemos un *InputObject* para el ratón en las líneas 17-18 (que eliminaremos en el destructor en 23).

El método *frameStarted* es algo más complejo que el estudiado en el ejemplo anterior. En esta ocasión, se definen en las líneas 28-30 una serie de variables que comentaremos a continuación. En `vt` almacenaremos el vector de traslación relativo que aplicaremos a la cámara, dependiendo de la pulsación de teclas del usuario. La variable `r` almacenará la rotación que aplicaremos al nodo de la escena (pasado como tercer argumento al constructor). En `deltaT` guardaremos el número de segundos transcurridos desde el despliegue del último frame. Finalmente la variable `tSpeed` servirá para indicar la traslación (distancia en unidades del mundo) que queremos recorrer con la cámara en un segundo.

La necesidad de medir el tiempo transcurrido desde el despliegue del último frame es imprescindible si queremos que las unidades del espacio recorridas por el modelo (o la cámara en este caso) sean dependientes del tiempo e independientes de las capacidades de representación gráficas de la máquina sobre la que se están ejecutando. Si aplicamos directamente un incremento del espacio sin tener en cuenta el tiempo, como se muestra en la Figura 13.3, el resultado del espacio recorrido dependerá de la velocidad de despliegue (ordenadores más potentes avanzarán más espacio). La solución es aplicar incrementos en la distancia dependientes del tiempo.

De este modo, aplicamos un movimiento relativo a la cámara (en función de sus ejes locales), multiplicando el vector de traslación (que ha sido definido dependiendo de la pulsación de teclas del usuario en las líneas 34-37), por `deltaT` y por la velocidad de traslación (de modo que avanzará 20 unidades por segundo).

De forma similar, cuando el usuario pulse la tecla R, se rotará el modelo respecto de su eje Y local a una velocidad de 180^o por segundo (líneas 40-41).

Finalmente, se aplicará una rotación a la cámara respecto de sus ejes Y y Z locales empelando el movimiento relativo del ratón. Para ello, se obtiene el estado del ratón (líneas 44-45), obteniendo el incremento relativo en píxeles desde la última captura (mediante el atributo `abs` se puede obtener el valor absoluto del incremento).

13.3. Creación manual de Entidades

Ogre soporta la creación manual de multitud de entidades y objetos. Veremos a continuación cómo se pueden añadir planos y fuentes de luz a la escena.

La creación manual del plano se realiza en las líneas [8-11] del siguiente listado. En la línea [8] se especifica que el plano tendrá como vector normal, el vector unitario en Y, y que estará situado a -5 unidades respecto del vector normal. Esta definición se corresponde con un plano infinito (descripción matemática abstracta). Si queremos representar el plano, tendremos que indicar a Ogre el tamaño (finito) del mismo, así como la resolución que queremos aplicarle (número de divisiones horizontales y verticales). Estas operaciones se indican en las líneas [9-11].

Listado 13.9: Definición de createScene (MyApp.cpp)

```
1  void MyApp::createScene() {
2    Ogre::Entity* ent1 = _sceneManager->createEntity("Sinbad.mesh");
3    Ogre::SceneNode* node1 =
4          _sceneManager->createSceneNode("SinbadNode");
5    node1->attachObject(ent1);
6    _sceneManager->getRootSceneNode()->addChild(node1);
7
8    Ogre::Plane pl1(Ogre::Vector3::UNIT_Y, -5);
9    Ogre::MeshManager::getSingleton().createPlane("pl1",
10     Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
11     pl1,200,200,1,1,true,1,20,20,Ogre::Vector3::UNIT_Z);
12
13   Ogre::SceneNode* node2 = _sceneManager->createSceneNode("node2");
14   Ogre::Entity* grEnt = _sceneManager->createEntity("pEnt", "pl1");
15   grEnt->setMaterialName("Ground");
16   node2->attachObject(grEnt);
17
18   _sceneManager->setShadowTechnique(Ogre::
         SHADOWTYPE_STENCIL_ADDITIVE);
19   Ogre::Light* light = _sceneManager->createLight("Light1");
20   light->setType(Ogre::Light::LT_DIRECTIONAL);
21   light->setDirection(Ogre::Vector3(1,-1,0));
22   node2->attachObject(light);
23
24   _sceneManager->getRootSceneNode()->addChild(node2);
25 }
```

El uso del *MeshManager* nos permite la creación de planos, skyboxes, superficies de Bezier, y un largo etcétera. En el caso de este ejemplo, se utiliza la llamada a `createPlane` que recibe los siguientes parámetros indicados en orden: el primer parámetro es el nombre de la malla resultante, a continuación el nombre del grupo de mallas (emplearemos el grupo por defecto). El tercer parámetro es el objeto definición del plano infinito (creado en la línea [8]). Los siguientes dos parámetros indican el ancho y alto del plano en coordenadas del mundo (200x200 en este caso). Los siguientes dos parámetros se corresponden con el número de segmentos empleados para definir el plano (1x1 en este caso), como se indica en la Figura 13.4. Estos parámetros sirven para especificar la resolución geométrica del plano creado (por si posteriormente queremos realizar operaciones de distorsión a nivel de vértice, por ejemplo).

El siguiente parámetro booleano indica (si es cierto) que los vectores normales se calcularán perpendiculares al plano. El siguiente parámetro (por defecto, 1) indica el conjunto de coordenadas de textura que serán creadas. A continuación se indica el número de repetición (*uTile* y *vTile*) de la textura (ver Figura 13.6). El último parámetro indica la dirección del vector *Up* del plano.

A continuación en la línea [15] asignamos al plano el material "*Ground*". Este material está definido en un archivo en el directorio `media` que ha sido cargado empleando el cargador de recursos explicado en la sección 13.1.2. Este material permite la recepción de sombras, tiene componente de brillo difuso y especular, y una textura de imagen basada en el archivo *ground.jpg*. Estudiaremos en detalle la asignación de materiales en el próximo capítulo del documento.

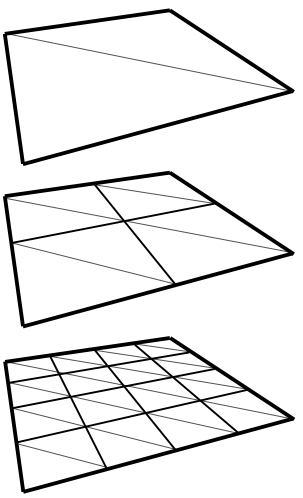


Figura 13.4: Número de segmentos de definición del plano. La imagen superior implica un segmento en X y un segmento en Y (valores por defecto). La imagen central se corresponde con una definición de 2 segmentos en ambos planos, y la inferior de 4 segmentos.

```
material Ground
{
 receive_shadows on
 technique
 {
   pass
   {
     ambient 0.7 0.7 0.7
     diffuse 1 1 1 1
     texture_unit
     {
       texture
         ground.jpg
     }
   }
 }
}
```

Figura 13.5: Fichero de definición del material `Ground.material`.

Finalmente las líneas 18-22 se encargan de definir una fuente de luz dinámica direccional [2] y habilitar el cálculo de sombras. Ogre soporta 3 tipos básicos de fuentes de luz y 11 modos de cálculo de sombras dinámicas, que serán estudiados en detalle en próximos capítulos.

13.4. Uso de Overlays

En el último ejemplo del capítulo definiremos superposiciones (*Overlays*) para desplegar en 2D elementos de información (tales como marcadores, botones, logotipos, etc...).

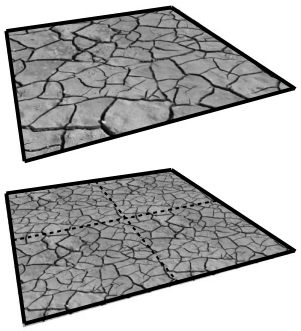


Figura 13.6: Valores de repetición en la textura. La imagen superior se corresponde con valores de $uTile = 1, vTile = 1$. La imagen inferior implica un valor de 2 en ambos parámetros (repitiendo la textura 2x2 veces en el plano).

```
Blue
{
  type truetype
  source Blue.ttf
  size 25
  resolution 96
}
```

Figura 13.7: Fichero de definición de la fuente `Blue.fontdef`.

La gestión de los *Overlays* en Ogre se realiza mediante el gestor de recursos llamado *OverlayManager*, que se encarga de cargar los scripts de definición de los *Overlays*. En un overlay pueden existir objetos de dos tipos: los **contenedores** y los **elementos**. Los elementos pueden ser de tres tipos básicos: *TextArea* empleado para incluir texto, *Panel* empleado como elemento que puede agrupar otros elementos con un fondo fijo y *BorderPanel* que es igual que *Panel* pero permite que la textura de fondo se repita y definir un borde independiente del fondo. La definición del *Overlay* del ejemplo de esta sección se muestra en la Figura 13.8. En el caso de utilizar un *TextArea*, será necesario especificar la carga de una fuente de texto. Ogre soporta texturas basadas en imagen o truetype. La definición de las fuentes se realiza empleando un fichero de extensión `.fontdef` (ver Figura 13.7).

En el listado de la Figura 13.8 se muestra la definición de los elementos y contendores del Overlay, así como los ficheros de material (extensión `.material`) asociados al mismo. En el siguiente capítulo estudiaremos en detalle la definición de materiales en Ogre, aunque un simple vistazo al fichero de materiales nos adelanta que estamos creando dos materiales, uno llamado *panelInfoM*, y otro *matUCLM*. Ambos utilizan texturas de imagen. El primero de ellos utiliza una técnica de mezclado que tiene en cuenta la componente de opacidad de la pasada.

El primer contenedor de tipo Panel (llamado *PanelInfo*), contiene cuatro TextArea en su interior. El posicionamiento de estos elementos se realiza de forma relativa al posicionamiento del contenedor. De este modo, se establece una relación de jerarquía implícita entre el contenedor y los objetos contenidos. El segundo elemento contenedor (llamado *logoUCLM*) no tiene ningún elemento asociado en su interior.

La Tabla 13.1 describe los principales atributos de los contenedores y elementos de los overlays. Estas propiedades pueden ser igualmente modificadas en tiempo de ejecución, por lo que son *animables*. Por ejemplo, es posible modificar la rotación de un elemento del *Overlay* consiguiendo bonitos efectos en los menús del juego.

El sistema de scripting de Overlays de Ogre permite definir plantillas de estilo que pueden utilizar los elementos y contenedores. En este ejemplo se han definido dos plantillas, una para texto genérico (de nombre *MyTemplates/Text*), y otra para texto pequeño (*MyTemplates/SmallText)*. Para aplicar estas plantillas a un elemento del *Overlay*, basta con indicar seguido de dos puntos el nombre de la plantilla a continuación del elemento (ver Figura 13.8).

En el listado del Overlay definido, se ha trabajado con el modo de especificación del tamaño en píxeles. La alineación de los elementos se realiza a nivel de píxel (en lugar de ser relativo a la resolución de la pantalla), obteniendo un resultado como se muestra en la Figura 13.8. El uso de valores negativos en los campos de alineación permite posicionar con exactitud los paneles alineados a la derecha y en el borde infe-

[2] Este tipo de fuentes de luz únicamente tienen dirección. Definen rayos de luz paralelos de una fuente distante. Es el equivalente al tipo de fuente *Sun* de Blender.

info.overlay

```
template element TextArea(MyTemplates/Text)
{
  font_name Blue
  metrics_mode pixels
  char_height 15
  colour 1.0 1.0 1.0
}

template element TextArea(MyTemplates/SmallText)
{
  font_name Blue
  metrics_mode pixels
  char_height 12
  colour 1.0 1.0 1.0
}
```

panel.material

```
material panelInfoM
{
 technique
 {
   pass
   {
     scene_blend src_alpha one_minus_src_alpha
     texture_unit
     {
       texture panel.tga
     }
   }
 }
}

material matUCLM
{
 technique
 {
   pass
   {
     texture_unit
     {
       texture logouclm.jpg
     }
   }
 }
}
```

```
Info
{
  zorder 500
  container Panel(panelInfo)
  {
    metrics_mode pixels
    left 10
    top -140
    width 280
    height 130
    vert_align bottom
    material panelInfoM

    element TextArea(fpsInfo) : MyTemplates/Text
    {
      top 35
      left 180
    }

    element TextArea(camPosInfo) : MyTemplates/Text
    {
      top 65
      left 120
    }

    element TextArea(camRotInfo) : MyTemplates/SmallText
    {
      top 80
      left 120
    }

    element TextArea(modRotInfo) : MyTemplates/Text
    {
      top 105
      left 120
    }
  }

  container Panel(logoUCLM)
  {
    metrics_mode pixels
    left -180
    top 0
    width 150
    height 120
    vert_align top
    horz_align right
    material matUCLM
  }
}
```

Figura 13.8: Definición de los Overlays empleando ficheros externos. En este fichero se describen dos paneles contenedores; uno para la zona inferior izquierda que contiene cuatro áreas de texto, y uno para la zona superior derecha que representará el logotipo de la UCLM.

rior. Por ejemplo, en el caso del panel *logoUCLM*, de tamaño (150x120), la alineación horizontal se realiza a la derecha. El valor de -180 en el campo *left* indica que queremos que quede un margen de 30 píxeles entre el lado derecho de la ventana y el extremo del logo.

Los *Overlays* tienen asociada una profundidad, definida en el campo *zorder* (ver Figura 13.8), en el rango de 0 a 650. *Overlays* con menor valor de este campo serán dibujados encima del resto. Esto nos permite definir diferentes niveles de despliegue de elementos 2D.

Para finalizar estudiaremos la modificación en el código de *MyApp* y *MyFrameListener*. En el siguiente listado se muestra que el constructor del *FrameListener* (línea 7) necesita conocer el *OverlayManager*, cuya referencia se obtiene en la línea 14. En realidad no sería necesario pasar el puntero al constructor, pero evitamos de esta forma que el *FrameListener* tenga que solicitar la referencia.

Atributo	Descripción
metrics_mode	Puede ser *relative* (valor entre 0.0 y 1.0), o *pixels* (valor en píxeles). Por defecto es *relative*. En coordenadas relativas, la coordenada superior izquierda es la (0,0) y la inferior derecha la (1,1).
horz_align	Puede ser *left* (por defecto), *center* o *right*.
vert_align	Puede ser *top* (por defecto), *center* o *bottom*.
left	Posición con respecto al extremo izquierdo. Por defecto 0. Si el valor es negativo, se interpreta como espacio desde el extremo derecho (con alineación *right*).
top	Posición con respecto al extremo superior. Por defecto 0. Si el valor es negativo, se interpreta como espacio desde el extremo inferior (con alineación vertical *bottom*).
width	Ancho. Por defecto 1 (en *relative*).
height	Alto. Por defecto 1 (en *relative*).
material	Nombre del material asociado (por defecto Ninguno).
caption	Etiqueta de texto asociada al elemento (por defecto Ninguna).
rotation	Ángulo de rotación del elemento (por defecto sin rotación).

Cuadro 13.1: Atributos generales de *Elementos* y *Contenedores* de Overlays.

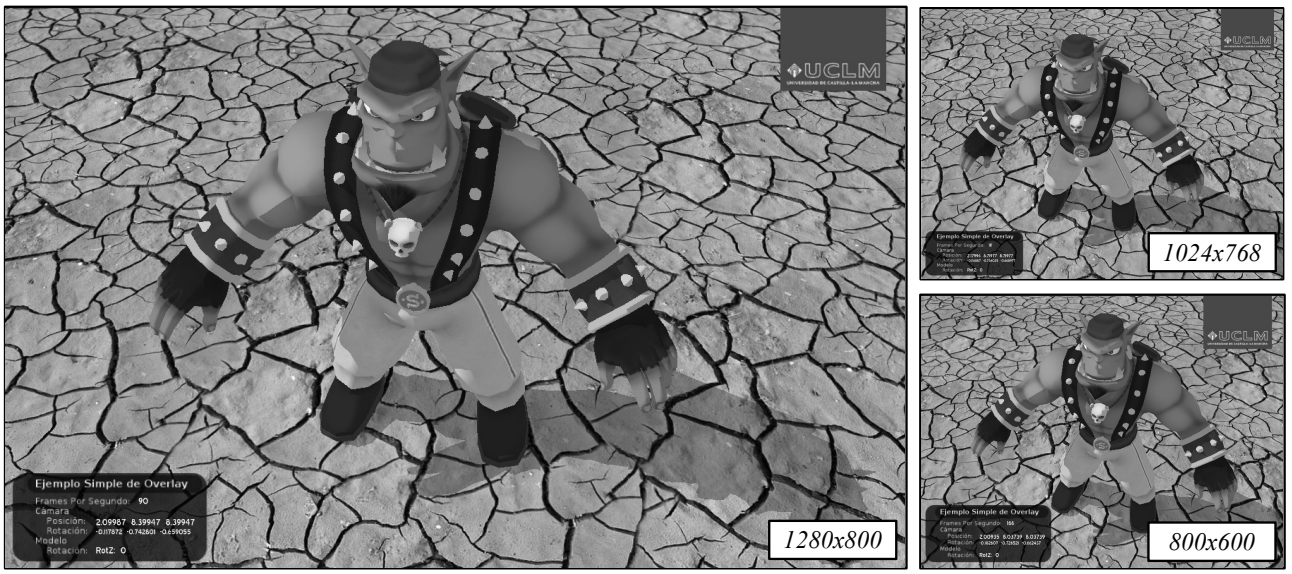


Figura 13.9: Resultado de ejecución del ejemplo de uso de Overlays, combinado con los resultados parciales de las secciones anteriores. La imagen ha sido generada con 3 configuraciones de resolución diferentes (incluso con diferente relación de aspecto): 1280x800, 1024x768 y 800x600.

En las líneas 15-16, obtenemos el *Overlay* llamado *Info* (definido en el listado de la Figura 13.8), y lo mostramos.

Para finalizar, el código del *FrameListener* definido en las líneas 6-16 del siguiente listado parcial, modifica el valor del texto asociado a los elementos de tipo TextArea. Hacemos uso de la clase auxiliar *Ogre::StringConverter* que facilita la conversión a String de ciertos tipos de datos (vectores, cuaternios...).

Listado 13.10: Modificación en MyApp.cpp

```
1 int MyApp::start() {
2     ...
```

```
3   loadResources();
4   createScene();
5   createOverlay();   // Metodo propio para crear el overlay
6    ...
7   _framelistener = new MyFrameListener(window, cam, node,
8                        _overlayManager);
9   _root->addFrameListener(_framelistener);
10   ...
11 }
12
13 void MyApp::createOverlay() {
14   _overlayManager = Ogre::OverlayManager::getSingletonPtr();
15   Ogre::Overlay *overlay = _overlayManager->getByName("Info");
16   overlay->show();
17 }
```

Listado 13.11: Modificación en MyFrameListener.cpp

```
1  bool MyFrameListener::frameStarted(const Ogre::FrameEvent& evt) {
2    ...
3    _camera->yaw(Ogre::Radian(rotx));
4    _camera->pitch(Ogre::Radian(roty));
5
6    Ogre::OverlayElement *oe;
7    oe = _overlayManager->getOverlayElement("fpsInfo");
8    oe->setCaption(Ogre::StringConverter::toString(fps));
9    oe = _overlayManager->getOverlayElement("camPosInfo");
10   oe->setCaption(Ogre::StringConverter::toString(_camera->
         getPosition()));
11   oe = _overlayManager->getOverlayElement("camRotInfo");
12   oe->setCaption(Ogre::StringConverter::toString(_camera->
         getDirection()));
13   oe = _overlayManager->getOverlayElement("modRotInfo");
14   Ogre::Quaternion q = _node->getOrientation();
15   oe->setCaption(Ogre::String("RotZ: ") +
16      Ogre::StringConverter::toString(q.getYaw()));
17
18   return true;
19 }
```

Capítulo 14

Interacción y Widgets

César Mora Castro

Como se ha visto a lo largo del curso, el desarrollo de un videojuego requiere tener en cuenta una gran variedad de disciplinas y aspectos: gráficos 3D o 2D, música, simulación física, efectos de sonido, jugabilidad, eficiencia, etc. Uno de los aspectos a los que se les suele dar menos importancia, pero que juegan un papel fundamental a la hora de que un juego tenga éxito o fracase, es la *interfaz de usuario*. Sin embargo, el mayor inconveniente es que la mayoría de los motores gráficos no dan soporte para la gestión de *Widgets*, y realizarlos desde cero es un trabajo más costoso de lo que pueda parecer.

En este capítulo se describe de forma general la estructura y las guías que hay que tener en cuenta a la hora de diseñar y desarrollar una interfaz de usuario. Además, se explicará el uso de *CEGUI*, una biblioteca de gestión de *Widgets* para integrarlos en motores gráficos.

Eficiencia y diseño

Para desarrollar un videojuego, tan importante es cuidar la eficiencia y optimizarlo, como que tenga un diseño visualmente atractivo.

14.1. Interfaces de usuario en videojuegos

Las interfaces de usuario específicas de los videojuegos deben tener el objetivo de crear una sensación positiva, que consiga la mayor inmersión del usuario posible. Estas interfaces deben tener lo que se denomina *flow*. El *flow*[50] es la capacidad de atraer la atención del usuario, manteniendo su concentración, la inmersión dentro de la trama del videojuego, y que consiga producir una experiencia satisfactoria.

Cada vez se realizan estudios más serios sobre cómo desarrollar interfaces que tengan *flow*, y sean capaces de brindar una mejor experiencia al usuario. La principal diferencia con las interfaces de usuario de aplicaciones no orientadas al ocio, es que estas centran su diseño en la usabilidad y la eficiencia, no en el impacto *sensorial*. Si un videojuego no consigue atraer y provocar sensaciones positivas al usuario, este posiblemente no triunfará, por muy eficiente que sea, o por muy original o interesante que sea su trama.

A continuación se detalla una lista de aspectos a tener en cuenta que son recomendables para aumentar el *flow* de un videojuego, aunque tradicionalmente se han considerado *perjudiciales* a la hora de diseñar interfaces tradicionales según las reglas de interacción persona-computador:

- *Mostrar la menor cantidad de información posible*: durante el transcurso del juego, es mejor no sobrecargar la interfaz con una gran cantidad de información. En la gran mayoría de videojuegos, toda esta configuración se establece *antes* de comenzar el juego (por ejemplo, en el *Menú*), por lo que la interfaz queda menos sobrecargada y distrae menos al usuario. Incluso el usuario puede tener la opción de mostrar menos información aún si lo desea.

- *Inconsistencia de acciones:* en algunos casos, es posible que se den inconsistencias en las acciones dependiendo del contexto del personaje. Por ejemplo, el botón de saltar cuando el personaje está en tierra puede ser el de nadar si de repente salta al agua.

  Es importante mantener un número reducido de teclas (tanto si es por limitaciones de la plataforma, como una videoconsola, como para hacer la usabilidad más sencilla al usuario). Por lo tanto, hay que conseguir agrupar las acciones en los botones según su naturaleza. Por ejemplo, un botón lleva a cabo acciones con objetos y peronajes (hablar, abrir una puerta), y otro movimientos de desplazamiento (saltar, escalar). Esto aumentará la intuitividad y la usabilidad del videojuego, y por lo tanto, su *flow*.

- *Dificultar los objetivos al usuario:* una de las reglas de oro de la interacción persona-computador es prevenir al usuario de cometer errores. Sin embargo, en los videojuegos esta regla puede volverse contradictoria, pues en la mayoría de los casos el usuario busca en los videojuegos un sentimiento de satisfacción que se logra por medio de la superación de obstáculos y desafíos.

  Por lo tanto, es también de vital importancia conseguir un equilibrio en la dificultad del juego, que no sea tan difícil como para frustrar al usuario, pero no tan fácil como para que resulte aburrido. En este aspecto la interfaz juega un papel de mucho peso.

Se ha visto cómo el caso particular de las interfaces de los videojuegos puede contradecir reglas que se aplican en el diseño de interfaces de usuario clásicas en el campo de la interacción persona-computador. Sin embargo, existen muchas recomendaciones que son aplicables a ambos tipos de interfaces. A continuación se explican algunas:

- *Mantener una organización intuitiva:* es importante que el diseño de los *menús* sean intuitivos. En muchos casos, esta falta de organización crea confusión innecesaria al usuario.

- *Ofrecer una legibilidad adecuada:* en algunos casos, darle demasiada importancia a la estética puede implicar que la legibilidad del texto de las opciones o botones se vea drásticamente reducida. Es importante mantener la funcionalidad básica.

- *Esperas innecesarias:* en multitud de videojuegos, el usuario se ve forzado a esperar a que una determinada película o animación se reproduzca de forma completa, sin poder omitirla. Incluso en el caso en que pueda omitirse, el usuario ha debido esperar previamente a la carga del clip de vídeo para después poder omitirla. Este tipo de inconvenientes reduce notablemente el *flow* de la aplicación.

  Existen multitud de formas en las que el usuario realiza esperas innecesarias, y que aumenta su frustración. Por ejemplo, si quiere volver a repetir una acción, tiene que volver a confirmar uno por uno todos los parámetros, aunque estos no cambien. Este es el caso de los juegos de carreras, en el que para volver a repetir una carrera es necesario presionar multitud botones e incluso esperar varios minutos.

- *Ayuda en línea:* muchas veces el usuario necesita consultar el manual durante el juego para entender alguna característica de él. Sin embargo, este manual es documentación extensa que no rompe con la inmersión del videojuego. Es conveniente que se proporcione una versión *suave*, acorde con la estética y que muestre la información estrictamente necesaria.

En general, es importante tener en cuenta cuatro puntos importantes:

1. **Intuitividad:** cuán fácil es aprender a usar una interfaz de un videojuego.
2. **Eficiencia:** cuán rápido se puede realizar una tarea, sobretodo si es muy repetitiva.
3. **Simplicidad:** mantener los controles y la información lo más minimalista posible.
4. **Estética:** cuán sensorialmente atractiva es la interfaz.

Una vez vistas algunas guías para desarrollar interfaces de usuario de videojuegos atractivas y con *flow*, se va a describir la estructura básica que deben tener.

Existe una estructura básica que un videojuego debe seguir. No es buena práctica comenzar el juego directamente en el "terreno de juego" o "campo de batalla". La estructura típica que debe seguir la interfaz de un videojuego es la siguiente:

En primer lugar, es buena práctica mostrar una *splash screen*. Este tipo de pantallas se muestran al ejecutar el juego, o mientras se carga algún recurso que puede durar un tiempo considerable, y pueden ser usadas para mostrar información sobre el juego o sobre sus desarrolladores. Suelen mostrarse a pantalla completa, o de menor tamaño pero centradas. La Figura 14.1 muestra la *splash screen* del juego *free orion*.

Figura 14.1: Ejemplo de *Splash Screen* durante la carga del juego *FreeOrion*.

Las otros dos elementos de la esctructura de un videojuego son el Menú y el HUD. Suponen una parte muy importante de la interfaz de un juego, y es muy común utilizar *Widgets* en ellos. A continuacion se analizarán más en detalle y se mostrarán algunos ejemplos.

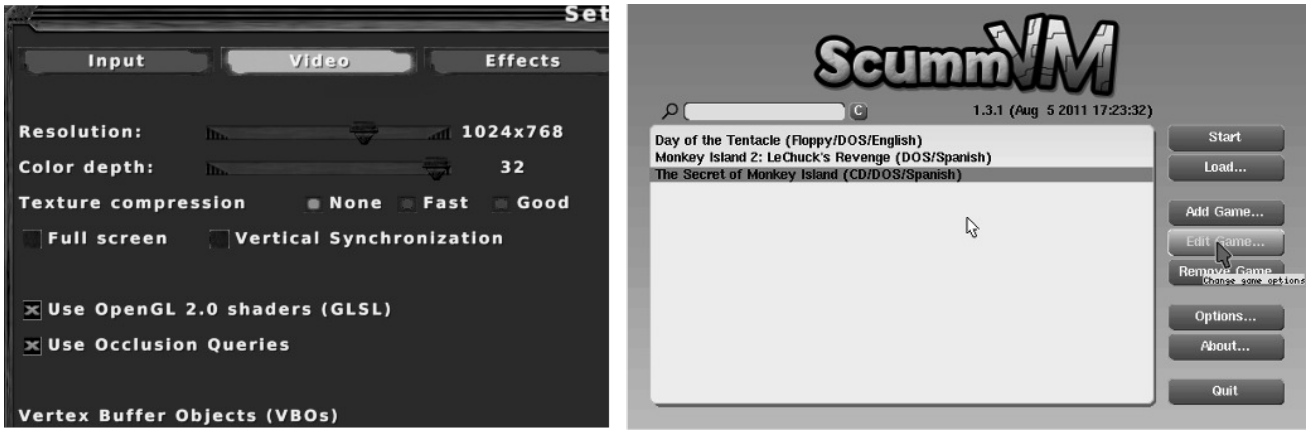


Figura 14.2: Extracto del menú de configuración de Nexuiz (*izquierda*), e interfaz de ScummVM (*derecha*).

14.1.1. Menú

Todos los videojuegos deben tener un *Menú* desde el cual poder elegir los modos de juego, configurar opciones, mostrar información adicional y otras características. Dentro de estos menús, es muy frecuente el uso de *Widgets* como botones, barras deslizantes (por ejemplo, para configurar la resolución de pantalla), listas desplegables (para elegir idioma), o *check buttons* (para activar o desactivar opciones). Por eso es importante disponer de un buen repertorio de *Widgets*, y que sea altamente personalizable para poder adaptarlos al estilo visual del videojuego.

En la Figura 14.2 se puede apreciar ejemplos de interfaces de dos conocidos juegos *open-source*. La interfaz de la izquierda, correspondiente al juego *Nexuiz*, muestra un trozo de su dialogo de configuración, mientras que la de la derecha corresponde a la interfaz de *ScummVM*. En estos pequeños ejemplos se muestra un número considerable de *Widgets*, cuyo uso es muy común:

- *Pestañas:* para dividir las opciones por categorías.
- *Barras de desplazamiento:* para configurar opciones que pueden tomar valores muy numerosos.
- *Radio buttons:* parámetros que pueden tomar valores excluyentes entre ellos.
- *Check buttons:* activan o desactivan una opción.

14.1.2. HUD

En relación a las interfaces de los videojuegos, concretamente se denomina *HUD* (del inglés, *Head-Up Display*), a la información y elementos de interacción mostrados durante el propio transcurso de la partida. La información que suelen proporcionar son la vida de los personajes, mapas, velocidad de los vehículos, etc.

En la Figura 14.3 se muestra una parte de la interfaz del juego de estrategia ambientada en el espacio *FreeOrion*. La interfaz utiliza elementos para mostrar los parámetros del juego, y también utiliza *Widgets* para interactuar con él.

Como se puede intuir, el uso de estos *Widgets* es muy común en cualquier videojuego, independientemente de su complejidad. Sin embargo, crear desde cero un conjunto medianamente funcional de estos es una tarea nada trivial, y que roba mucho tiempo de la línea principal de trabajo, que es el desarrollo del propio videojuego.

Una vez que se ha dado unas guías de estilo y la estructura básicas para cualquier videojuego, y se ha mostrado la importancia de los *Widgets* en ejemplos reales, se va a estudiar el uso de una potente biblioteca que proporciona estos mismos elementos para distintos motores gráficos, para que el desarrollo de *Widgets* para videojuegos sea lo menos problemático posible.

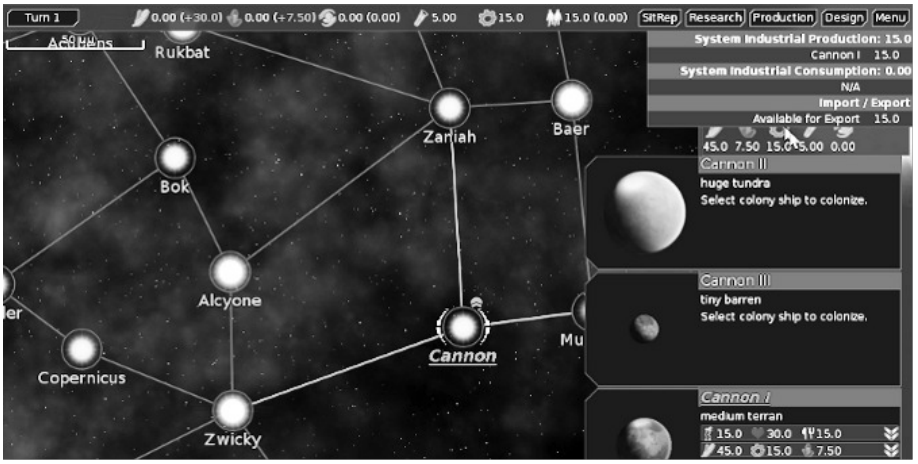


Figura 14.3: Screenshot del HUD del juego *FreeOrion*.

CEGUI's mission

Como dice en su página web, "CEGUI está dirigido a desarrolladores de videojuegos que deben invertir su tiempo en desarrollar buenos juegos, no creando subsistemas de interfaces de usuario".

14.2. Introducción CEGUI

CEGUI (Crazy Eddie's GUI)[2] (*Crazy Eddie's GUI*) es una biblioteca *open source* multiplataforma que proporciona entorno de ventanas y *Widgets* para motores gráficos, en los cuales no se da soporte nativo, o es muy deficiente. Es orientada a objetos y está escrita en C++.

CEGUI es una biblioteca muy potente en pleno desarrollo, por lo que está sujeta a continuos cambios. Todas las características y ejemplos descritos a lo largo de este capítulo se han creado utilizando la versión actualmente estable, la **0.7.x**. No se asegura el correcto funcionamiento en versiones anteriores o posteriores.

CEGUI y Ogre3D

A lo largo de estos capítulos, se utilizará la fórmula CEGUI/Ogre3D/OIS para proveer interfaz gráfica, motor de rendering y gestión de eventos a los videojuegos, aunque también es posible utilizarlo con otras bibliotecas (ver documentación de CEGUI).

CEGUI es muy potente y flexible. Es compatible con los motores gráficos OpenGL, Direct3D, Irrlicht y Ogre3D.

De la misma forma que Ogre3D es únicamente un motor de rendering, CEGUI es sólo un motor de de gestión de *Widgets*, por lo que el renderizado y la gestión de eventos de entrada deben ser realizadas por bibliotecas externas.

En sucesivas secciones se explicará cómo integrar CEGUI con las aplicaciones de este curso que hacen uso de Ogre3D y OIS. Además se mostrarán ejemplos prácticos de las características más importantes que ofrece.

14.2.1. Instalación

En las distribuciones actuales más comunes de GNU/Linux (Debian, Ubuntu), está disponible los paquetes de CEGUI para descargar, sin embargo estos dependen de la versión 1.7 de Ogre, por lo que de momento no es posible utilizar la combinación CEGUI+OGRE 1.8 desde los respositorios. Para ello, es necesario descargarse el código fuente de la última versión de CEGUI y compilarla. A continuación se muestran los pasos.

Descargar la última versión estable (0.7.7) desde la página web (*www.cegui.org.uk*), ir a la sección *Downloads* (columna de la izquierda). Pinchar sobre la versión 0.7.7, y en la sección *CEGUI 0.7.7 Library Source Downloads* descargar la versión para GNU/Linux.

Una vez descomprimido el código fuente, para compilar e instalar la biblioteca se ejecutan los siguientes comandos:

```
./configure cegui_enable_ogre=yes
make && sudo make install
```

Después de ejecutar el `./configure`, es importante asegurarse de que en el resumen se muestre, bajo la sección *Renderer Modules*, la opción *Building Ogre Renderer: yes*.

Estos pasos instalarán CEGUI bajo el directorio */usr/local/*, por lo que es importante indicar en el *Makefile* que las cabeceras las busque en el directorio */usr/local/include/CEGUI*.
Además, algunos sistemas operativos no buscan las bibliotecas de enlazado dinámico en /usr/local/lib por defecto. Esto produce un error al ejecutar la aplicación indicando que no puede encontrar las bibliotecas libCEGUI*. Para solucionarlo, se puede editar como superusuario el fichero */etc/ld.so.conf*, y añadir la línea *include /usr/local/lib*. Para que los cambios surtan efecto, ejecutar `sudo ldconfig`.

14.2.2. Inicialización

La arquitectura de CEGUI es muy parecida a la de Ogre3D, por lo que su uso es similar. Está muy orientado al uso de *scripts*, y hace uso del patrón *Singleton* para implementar los diferentes subsistemas. Los más importantes son:

- `CEGUI::System`: gestiona los parámetros y componentes más importantes de la biblioteca.
- `CEGUI::WindowManager`: se encarga de la creación y gestión de las *windows* de CEGUI.
- `CEGUI::SchemeManager`: gestiona los diferentes esquemas que utilizará la interfaz gráfica.
- `CEGUI::FontManager`: gestiona los distintos tipos de fuentes de la interfaz.

Estos subsistemas ofrecen funciones para poder gestionar los diferentes recursos que utiliza CEGUI. A continuación se listan estos tipos de recursos:

- *Schemes:* tienen la extensión *.scheme*. Definen el repertorio (o esquema) de *Widgets* que se utilizarán. También indica qué scripts utilizará de otros tipos, como por ejemplo el *ImageSet*, las *Fonts* o el *LookNFeel*.
- *Imageset:* tienen la extensión *.imageset*. Define cuáles serán la imágenes de los elementos de la interfaz (punteros, barras, botones, etc).
- *LookNFeel:* tienen la extensión *.looknfeel*. Define el *comportamiento visual* de cada uno de los *Widgets* para distintas acciones, por ejemplo, cómo se muestran cuando se pasa el puntero del ratón o cuando se presionan,
- *Fonts:* tienen la extensión *.font*. Cada uno de los scripts define un tipo de fuente junto con propiedades específicas como su tamaño.
- *Layouts:* tienen la extensión *.layout*. Cada script define clases de *ventanas* concretas, con cada uno de sus elementos. Por ejemplo, una ventana de chat o una consola.

Según se avance en el capítulo se irá estudiando más en profundidad cuál es el funcionamiento y la estructura de estos recursos.

En el siguiente código se muestran los primeros pasos para poder integrar CEGUI con Ogre3D, y de qué forma se inicializa.

Listado 14.1: Inicializacón de CEGUI para su uso con Ogre3D

```
1  #include <CEGUI.h>
2  #include <RendererModules/Ogre/CEGUIOgreRenderer.h>
3
4  CEGUI::OgreRenderer* renderer = &CEGUI::OgreRenderer::
       bootstrapSystem();
5
6  CEGUI::Scheme::setDefaultResourceGroup("Schemes");
7  CEGUI::Imageset::setDefaultResourceGroup("Imagesets");
8  CEGUI::Font::setDefaultResourceGroup("Fonts");
9  CEGUI::WindowManager::setDefaultResourceGroup("Layouts");
10 CEGUI::WidgetLookManager::setDefaultResourceGroup("LookNFeel");
```

Este método de inicializar el *renderer* utilizando el *bootstrapSystem* fue introducido a partir de la versión `0.7.1`. Para inicializar CEGUI en versiones anteriores, es necesario referirse a su documentacion.

En las líneas 1 y 2 se insertan las cabeceras necesarias. La primera incluye la biblioteca general, y en la segunda se indica de forma concreta que se va a utilizar el motor gráfico Ogre3D. En la línea 4 se inicializa CEGUI para ser utilizado con Ogre3D. Además es necesario indicar dónde estarán los recursos que utilizará la interfaz gráfica, tanto los scripts que utiliza, como las fuentes o las imágenes.

Dependiendo de la distribución que se utilice, estos recursos pueden venir con el paquete del repositorio o no. En este ejemplo vamos a considerar que debemos descargarlos aparte. Se pueden conseguir directamente descargando el código fuente de CEGUI desde su página[2]. Las distribuciones que los proporcionan, suelen situarlos en */usr/share/CEGUI/* o */usr/local/share/CEGUI/*

Para que CEGUI pueda encontrar los recursos, es necesario añadir estos grupos al fichero *resources.cfg* de Ogre.

Listado 14.2: Contenido del fichero *resources.cfg*

```
1  [General]
2  FileSystem=media
3  [Schemes]
4  FileSystem=media/schemes
5  [Imagesets]
6  FileSystem=media/imagesets
7  [Fonts]
8  FileSystem=media/fonts
9  [Layouts]
10 FileSystem=media/layouts
11 [LookNFeel]
12 FileSystem=media/looknfeel
```

Y las opciones que hay que añadir al *Makefile* para compilar son:

Listado 14.3: Flags de compilación y de enlazado de CEGUI.

```
1 #Flags de compilado
2 CXXFLAGS += `pkg-config --cflags CEGUI-OGRE`
3
4 #Flags de enlazado
5 LDFLAGS += `pkg-config --libs-only-L CEGUI-OGRE`
6 LDLIBS += `pkg-config --libs-only-l CEGUI-OGRE`
```

Es importante incluir los *flags* de compilado, en la línea 2, para que encuentre las cabeceras según se han indicado en el ejemplo de inicialización.

Esto es sólo el código que inicializa la biblioteca en la aplicación para que pueda comenzar a utilizar CEGUI como interfaz gráfica, todavía no tiene ninguna funcionalidad. Pero antes de empezar a añadir *Widgets*, es necesario conocer otros conceptos primordiales.

14.2.3. El Sistema de Dimensión Unificado

El posicionamiento y tamaño de los distintos *Widgets* no es tan trivial como indicar los valores absolutos. Puede ser deseable que un *Widget* se reposicione y redimensione si el *Widget* al que pertenece se redimensiona, por ejemplo.

CEGUI utiliza lo que denomina el Sistema de Dimensión Unificado (*Unified Dimension System*). El elemento principal de este sistema es:

CEGUI::UDim(scale, offset)

Indica la posición en una dimensión. El primer parámetro indica la posición relativa, que toma un valor entre 0 y 1, mientras que el segundo indica un desplazamiento absoluto en píxeles.

Por ejemplo, supongamos que posicionamos un *Widget* con *UDim* 0.5,20 en la dimensión *x*. Si el ancho de la pantalla fuese 640, la posición sería 0.5*640+20 = 340, mientras que si el ancho fuese 800, la posición seriá 0.5*800+20 = 420. En la Figura 14.5 se aprecian los dos ejemplos de forma gráfica. De esta forma, si la resolución de la pantalla cambia, por ejemplo, el *Widget* se reposicionará y se redimensionará de forma automática.

Teniendo en cuenta cómo expresar el posicionamiento en una única dimensión utilizando *UDim*, se definen dos elementoss más.

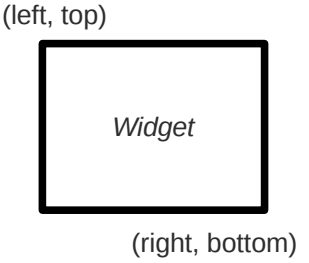


Figura 14.4: Área rectangular definida por *URect*.

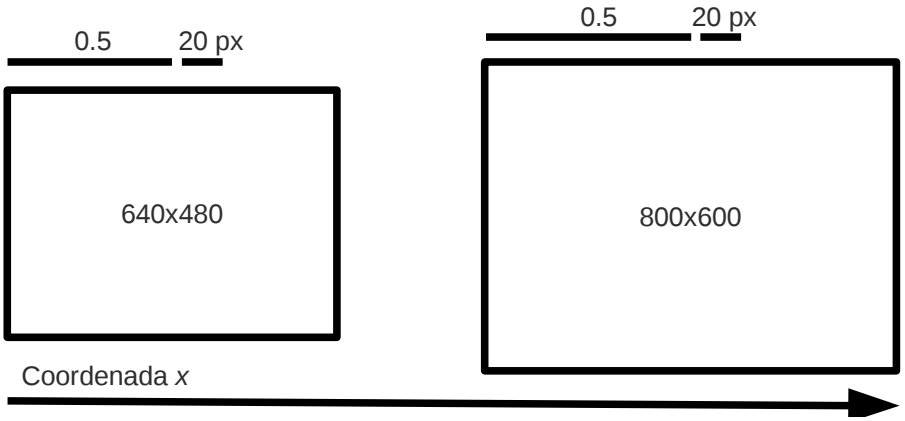


Figura 14.5: Ejemplo del funcionamiento de *UDim*.

Para definir un punto o el tamaño de un *Widget* se usa:

CEGUI::UVector2(UDim x, UDim y)

que está compuesto por dos *UDim*, uno para la coordenada *x*, y otro para la *y*, o para el ancho y el alto, dependiendo de su uso.

El segundo elemento, se utiliza para definir un área rectangular:

CEGUI::URect(UDim left, UDim top, UDim right, UDIM bottom)

Como muestra la Figura 14.4, los dos primeros definen la esquina superior izquierda, y los dos últimos la esquina inferior derecha.

14.2.4. Detección de eventos de entrada

Puesto que CEGUI es únicamente un motor de gestión de *Widgets*, tampoco incorpora la detección de eventos de entrada, por lo que es necesario *inyectárselos* desde otra biblioteca. En este caso, se aprovechará la que ya se ha estudiado: *OIS*.

Considerando que se utiliza *OIS* mediante *callbacks* (en modo *buffered*), hay que añadir las siguientes líneas para enviar a CEGUI la pulsación y liberación de teclas y de los botones del ratón.

Listado 14.4: Inyección de eventos de pulsación y liberación de teclas a CEGUI.

```
1  bool MyFrameListener::keyPressed(const OIS::KeyEvent& evt)
2  {
3    CEGUI::System::getSingleton().injectKeyDown(evt.key);
4    CEGUI::System::getSingleton().injectChar(evt.text);
5
6    return true;
7  }
8
9  bool MyFrameListener::keyReleased(const OIS::KeyEvent& evt)
10 {
11   CEGUI::System::getSingleton().injectKeyUp(evt.key);
12
13   return true;
```

```
14 }
15
16 bool MyFrameListener::mousePressed(const OIS::MouseEvent& evt, OIS
       ::MouseButtonID id)
17 {
18   CEGUI::System::getSingleton().injectMouseButtonDown(
         convertMouseButton(id));
19   return true;
20 }
21
22 bool MyFrameListener::mouseReleased(const OIS::MouseEvent& evt, OIS
       ::MouseButtonID id)
23 {
24   CEGUI::System::getSingleton().injectMouseButtonUp(
         convertMouseButton(id));
25   return true;
26 }
```

Además, es necesario convertir la forma en que identifica *OIS* los botones del ratón, a la que utiliza CEGUI, puesto que no es la misma, al contrario que sucede con las teclas del teclado. Para ello se ha escrito la función `convertMouseButton()`:

Listado 14.5: Función de conversión entre identificador de botones de ratón de OIS y CEGUI.

```
1 CEGUI::MouseButton MyFrameListener::convertMouseButton(OIS::
      MouseButtonID id)
2 {
3   CEGUI::MouseButton ceguiId;
4   switch(id)
5     {
6     case OIS::MB_Left:
7       ceguiId = CEGUI::LeftButton;
8       break;
9     case OIS::MB_Right:
10      ceguiId = CEGUI::RightButton;
11      break;
12    case OIS::MB_Middle:
13      ceguiId = CEGUI::MiddleButton;
14      break;
15    default:
16      ceguiId = CEGUI::LeftButton;
17    }
18  return ceguiId;
19 }
```

Por otro lado, también es necesario decir a CEGUI cuánto tiempo ha pasado desde la detección del último evento, por lo que hay que añadir la siguiente línea a la función `frameStarted()`:

Listado 14.6: Orden que indica a CEGUI el tiempo transcurrido entre eventos.

```
1 CEGUI::System::getSingleton().injectTimePulse(evt.
      timeSinceLastFrame)
```

Hasta ahora se ha visto el funcionamiento básico de *CEGUI*, los tipos básicos de scripts que define, la inicialización, el sistema de posicionamiento y dimensionado que utiliza e incluso como enviarle eventos de entrada. Una vez adquiridos estos conocimientos, es momento de crear la primera aplicación de Ogre que muestre un *Widget* con funcionalidad, como se describirá en la siguiente sección.

14.3. Primera aplicación

En esta sección se van a poner en práctica los primeros conceptos descritos para crear una primera aplicación. Esta aplicación tendrá toda la funcionalidad de Ogre (mostrando a Sinbad), y sobre él un botón para salir de la aplicación.

Es importante tener en cuenta que en CEGUI, **todos los elementos son *Windows***. Cada uno de los *Windows* puede contener a su vez otros *Windows*. De este modo, pueden darse situaciones raras como que un botón contenga a otro botón, pero que en la práctica no suceden.

Listado 14.7: Código de la función `createGUI()`.

```
1  void MyApp::createGUI()
2  {
3    renderer = &CEGUI::OgreRenderer::bootstrapSystem();
4    CEGUI::Scheme::setDefaultResourceGroup("Schemes");
5    CEGUI::Imageset::setDefaultResourceGroup("Imagesets");
6    CEGUI::Font::setDefaultResourceGroup("Fonts");
7    CEGUI::WindowManager::setDefaultResourceGroup("Layouts");
8    CEGUI::WidgetLookManager::setDefaultResourceGroup("LookNFeel");
9
10   CEGUI::SchemeManager::getSingleton().create("TaharezLook.scheme")
         ;
11   CEGUI::System::getSingleton().setDefaultFont("DejaVuSans-10");
12   CEGUI::System::getSingleton().setDefaultMouseCursor("TaharezLook"
         ,"MouseArrow");
13
14   //Creating GUI Sheet
15   CEGUI::Window* sheet = CEGUI::WindowManager::getSingleton().
         createWindow("DefaultWindow","Ex1/Sheet");
16
17   //Creating quit button
18   CEGUI::Window* quitButton = CEGUI::WindowManager::getSingleton().
         createWindow("TaharezLook/Button","Ex1/QuitButton");
19   quitButton->setText("Quit");
20   quitButton->setSize(CEGUI::UVector2(CEGUI::UDim(0.15,0),CEGUI::
         UDim(0.05,0)));
21   quitButton->setPosition(CEGUI::UVector2(CEGUI::UDim(0.5-0.15/2,0)
         ,CEGUI::UDim(0.2,0)));
22   quitButton->subscribeEvent(CEGUI::PushButton::EventClicked,
23                   CEGUI::Event::Subscriber(&MyFrameListener::quit,
24                                            _framelistener));
25   sheet->addChildWindow(quitButton);
26   CEGUI::System::getSingleton().setGUISheet(sheet);
27 }
```

La función `createGUI()` se encarga de la inicialización de la interfaz gráfica y de la creación de los elementos que contendrá. Como se explicó en la Sección 14.2.2, de las líneas 3-8 se indica que se quiere utilizar Ogre3D como motor de rendering, y se indica a CEGUI dónde están los distintos recursos.

En la línea 10 se crea el esquema que se va a utilizar. Se recuerda que un esquema definía el conjunto de *Widgets* que se podrán utilizar en la aplicación, junto a los tipos de letras, apariencia o comportamiento visual. Es como la elección del *tema* de la interfaz. El fichero de script *TaharezLook.scheme* debe de encontrarse en algún lugar del que CEGUI tenga constancia. Como se ha definido en el fichero *resources.cfg*, los

esquemas (*Schemes*) deben estar en *media/schemes*. Desde su página web se pueden descargar otros ejemplos, como el esquema *Vanilla*. Más adelante se analizará brevemente el contenido de estos scripts, para poder ajustar la interfaz gráfica a la estética del videojuego.

En las líneas 11 y 12 se definen algunos parámetros por defecto. En el primer caso, el tipo de letra predeterminada, y en el segundo el cursor, ambos elementos definidos en *TaharezLook.scheme*.

Los *Widgets* de la interfaz gráfica se organizan de forma jerárquica, de forma análoga al grafo de escena Ogre. Cada *Widget* (o *Window*) debe pertenecer a otro que lo contenga. De este modo, debe de haber un *Widget* "padre" o "raíz". Este *Widget* se conoce en CEGUI como *Sheet* (del inglés, *hoja*, refiriéndose a la hoja en blanco que contiene todos los elementos). Esta se crea en la línea 15. El primer parámetro indica el tipo del *Window*, que será un tipo genérico, *DefaultWindow*. El segundo es el nombre que se le da a ese elemento. Con *Ex1* se hace referencia a que es el primer ejemplo (*Example1*), y el segundo es el nombre del elemento.

Convenio de nombrado

CEGUI no exige que se siga ningún convenio de nombrado para sus elementos. Sin embargo, es altamente recomendable utilizar un convenio jerárquico, utilizando la barra "/" como separador.

El siguiente paso es crear el botón. En la línea 18 se llama al *WindowManager* para crear un *Window* (hay que recordar que en CEGUI todo es un *Window*). El primer parámetro indica el tipo, definido en el esquema escogido, en este caso un botón. El segundo es el nombre del elemento, para el cual se sigue el mismo convenio de nombrado que para el *Sheet*.

Después se indican algunos parámetros del botón. En la línea 19 se indica el texto del botón, utilizando la fuente predeterminada del sistema que se indicó en la inicialización.

En la línea 20 se indica el tamaño. Como se explicó en la Sección 14.2.3, para el tamaño se utiliza un *UVector2*, que contiene dos valores, ancho y alto. Por lo tanto, el ancho de este botón siempre será 0.15 del ancho del *Sheet*, y el alto será 0.05.

Para indicar la posición del *Widget*, se opta por centrarlo horizontalmente. Como se puede ver en la línea 21, para la posición en la dimensión x se indica la mitad del ancho del *Sheet*, 0.5, menos la mitad del ancho del *Widget*, es decir, 0.15/2 (el tamaño se acaba de indicar en la línea 20). Esto se debe a que el posicionamiento toma como punto de referencia la esquina superior izquierda del *Widget*. Para la posición en el eje y se ha optado por un 0.2 del alto del *Sheet*.

Hasta el momento se ha creado el *Sheet* que contendrá todo el conjunto de *Widgets*, un botón con el texto "Quit", y con un tamaño y apariencia determinado. Ahora se le va a asociar un comportamiento para cuando se pulse. Para asociar comportamientos, es necesario suscribir las funciones que implementan el comportamiento a los elementos. En la línea 22 se asocia ese comportamiento. El primer parámetro indica a qué tipo de acción se asocia el comportamiento, y en el segundo qué función se ejecuta. Existen una extensa lista de acciones a las que se pueden asociar comportamientos, como por ejemplo:

- MouseClicked
- MouseEnters
- MouseLeaves
- EventActivated
- EventTextChanged

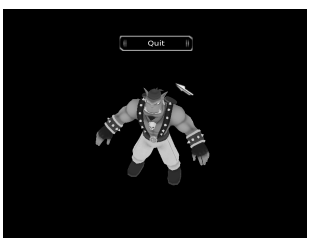


Figura 14.6: *Screenshot* de la primera aplicación de ejemplo.

- EventAlphaChanged
- EventSized

Como se ha estudiado anteriormente, al igual que pasa con el grafo de escena de Ogre3D, cada *Window* de CEGUI debe tener un padre. En la línea 25 se asocia el botón al *Sheet*, y por último, en la línea 26 se indica a CEGUI cuál es el *Sheet* que debe mostrar.

A continuación se muestra la definición de la función que implementa el comportamiento. Como se puede apreciar, simplemente cambia el valor de una variable booleana que controla la salida de la aplicación. Es importante tener en cuenta que no todas las funciones pueden ser utilizadas para implementar el comportamiento de los elementos. En su *signatura*, deben de tener como valor de retorno *bool*, y aceptar un único parámetro del tipo *const CEGUI::EventArgs& e*

Listado 14.8: Función que implementa el comportamiento del botón al ser pulsado.

```
1 bool MyFrameListener::quit(const CEGUI::EventArgs &e)
2 {
3   _quit = true;
4   return true;
5 }
```

En la Figura 14.6 se puede ver una captura de esta primera aplicación.

Ya que se ha visto cómo inicializar *CEGUI* y cuál es su funcionamiento básico, es momento de comenzar a crear interfaces más complejas, útiles, y atractivas.

14.4. Tipos de *Widgets*

Para comenzar a desarrollar una interfaz gráfica con CEGUI para un videojuego, primero es necesario saber cuál es exactamente el repertorio de *Widgets* disponible. Como se ha estudiado en secciones anteriores, el repertorio como tal está definido en el esquema escogido. Para los sucesivos ejemplos, vamos a utilizar el esquema *TaharezLook*, utilizado también en la primera aplicación de inicialización. CEGUI proporciona otros esquemas que ofrecen otros repertorios de *Widgets*, aunque los más comunes suelen estar implementados en todos ellos. Otros esquemas que proporciona CEGUI son *OgreTray*, *VanillaSkin* y *WindowsLook*.

El *script* del esquema define además la apariencia y el comportamiento visual de los *Widgets*. Para cambiar la apariencia visual, no es necesario crear un fichero esquema desde cero (lo que sería una ardua tarea). Basta con cambiar el *script* de los *ImageSet* y de los *Fonts*. Esto se estudiará con más profundidad en la Sección 14.8.

A continuación se muestra una pequeña lista de los *Widgets* más importantes definidos en el esquema *TaharezLook*:

- Button
- Check Box
- Combo Box
- Frame Window

- List Box
- Progress Bar
- Slider
- Static Text
- etc

El siguiente paso en el aprendizaje de CEGUI es crear una interfaz que bien podría servir para un juego, aprovechando las ventajas que proporcionan los *scripts*.

14.5. *Layouts*

Los *scripts* de layouts especifican qué *Widgets* habrá y su organización para una ventana específica. Por ejemplo, un *layout* llamado *chatBox.layout* puede contener la estructura de una ventana con un *editBox* para insertar texto, un *textBox* para mostrar la conversación, y un *button* para enviar el mensaje. De cada *layout* se pueden crear tantas instancias como se desee.

No hay que olvidar que estos ficheros son *xml*, por lo que deben seguir su estructura. La siguiente es la organización genérica de un recurso *layout*:

Listado 14.9: Estructura de un *script layout*.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <GUILayout>
3    <Window Type="WindowType" Name="Window1">
4      <Property Name="Property1" Value="Property1Value"/>
5      <Property Name="Property2" Value="Property2Value"/>
6      <!-- This is a comment -->
7      <Window Type="WindowType" Name="Window1/Window2">
8        <Property Name="Property1" Value="Property1Value"/>
9        <Property Name="Property2" Value="Property2Value"/>
10     </Window>
11     <!-- ... --!>
12   </Window>
13 </GUILayout>
```

En la línea 1 se escribe la cabecera del archivo *xml*, lo cual no tiene nada que ver con CEGUI. En la línea 2 se abre la etiqueta *GUILayout*, para indicar el tipo de *script* y se cierra en la línea 13.

A partir de aquí, se definen los *Windows* que contendrá la interfaz (¡en CEGUI todo es un *Window*!). Para declarar uno, se indica el tipo de *Window* y el nombre, como se puede ver en la línea 3. Dentro de él, se especifican sus propiedades (líneas 4 y 5). Estas propiedades pueden indicar el tamaño, la posición, el texto, la transparencia, etc. Los tipos de *Widgets* y sus propiedades se definían en el esquema escogido, por lo que es necesario consultar su documentación específica. En la Sección 14.6 se verá un ejemplo concreto.

En la línea 6 se muestra un comentario en *xml*.

Después de la definición de las propiedades de un *Window*, se pueden definir más *Windows* que pertenecerán al primero, ya que los *Widgets* siguen una estructura jerárquica.

14.6. Ejemplo de interfaz

El siguiente es el código de un *script layout* que define una ventana de configuración, con distintos *Widgets* para personalizar el volumen, la resolución, o el puerto para utilizar en modo *multiplayer*. Además incorpora un botón para aplicar los cambios y otro para salir de la aplicación. En la Figura 14.7 se muestra el resultado final.

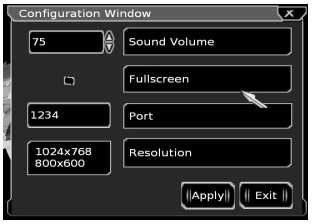


Figura 14.7: Resultado de la ventana de configuración del ejemplo.

Listado 14.10: Ejemplo de *layout* para crear una ventana de configuración.

```
<?xml version="1.0" encoding="UTF-8"?>
<GUILayout >
  <Window Type="TaharezLook/FrameWindow" Name="Cfg" >
    <Property Name="Text" Value="Cfguration Window" />
    <Property Name="TitlebarFont" Value="DejaVuSans-10" />
    <Property Name="TitlebarEnabled" Value="True" />
    <Property Name="UnifiedAreaRect" Value="
        {{0.133,0},{0.027,0},{0.320,300},{0.127,300}}" />
    <!-- Sonud parameter -->
    <Window Type="TaharezLook/StaticText" Name="Cfg/SndText" >
      <Property Name="Text" Value="Sonud Volume" />
      <Property Name="UnifiedAreaRect" Value="
          {{0.385,0},{0.0316,0},{0.965,0},{0.174,0}}" />
    </Window>
    <Window Type="TaharezLook/Spinner" Name="Cfg/SndVolume" >
      <Property Name="Text" Value="Sonud Volume" />
      <Property Name="StepSize" Value="1" />
      <Property Name="CurrentValue" Value="75" />
      <Property Name="MaximumValue" Value="100" />
      <Property Name="MinimumValue" Value="0" />
      <Property Name="UnifiedAreaRect" Value="
          {{0.0598,0},{0.046,0},{0.355,0},{0.166,0}}" />
    </Window>
    <!-- Fullscreen parameter -->
    <Window Type="TaharezLook/StaticText" Name="Cfg/FullScrText" >
      <Property Name="Text" Value="Fullscreen" />
      <Property Name="UnifiedAreaRect" Value="
          {{0.385,0},{0.226,0},{0.965,0},{0.367,0}}" />
    </Window>
    <Window Type="TaharezLook/Checkbox" Name="Cfg/FullscrCheckbox"
        >
      <Property Name="UnifiedAreaRect" Value="
          {{0.179,0},{0.244,0},{0.231,0},{0.370,0}}" />
    </Window>
    <!-- Port parameter -->
    <Window Type="TaharezLook/StaticText" Name="Cfg/PortText" >
      <Property Name="Text" Value="Port" />
      <Property Name="UnifiedAreaRect" Value="
          {{0.385,0},{0.420,0},{0.9656,0},{0.551,0}}" />
    </Window>
    <Window Type="TaharezLook/Editbox" Name="Cfg/PortEditbox" >
      <Property Name="Text" Value="1234" />
      <Property Name="MaxTextLength" Value="1073741823" />
      <Property Name="UnifiedAreaRect" Value="
          {{0.0541,0},{0.417,0},{0.341,0},{0.548,0}}" />
      <Property Name="TextParsingEnabled" Value="False" />
    </Window>
    <!-- Resolution parameter -->
    <Window Type="TaharezLook/StaticText" Name="Cfg/ResText" >
      <Property Name="Text" Value="Resolution" />
      <Property Name="UnifiedAreaRect" Value="
          {{0.385,0},{0.60,0},{0.965,0},{0.750,0}}" />
    </Window>
    <Window Type="TaharezLook/ItemListbox" Name="Cfg/ResListbox" >
      <Property Name="UnifiedAreaRect" Value="
          {{0.0530,0},{0.613,0},{0.341,0},{0.7904,0}}" />
      <Window Type="TaharezLook/ListboxItem" Name="Cfg/Res/Item1">
    <Property Name="Text" Value="1024x768"/>
      </Window>
```

```
50        <Window Type="TaharezLook/ListboxItem" Name="Cfg/Res/Item2">
51      <Property Name="Text" Value="800x600"/>
52        </Window>
53      </Window>
54      <!-- Exit button -->
55      <Window Type="TaharezLook/Button" Name="Cfg/ExitButton" >
56        <Property Name="Text" Value="Exit" />
57        <Property Name="UnifiedAreaRect" Value="
              {{0.784,0},{0.825,0},{0.968,0},{0.966,0}}" />
58      </Window>
59      <!-- Apply button -->
60      <Window Type="TaharezLook/Button" Name="Cfg/ApplyButton" >
61        <Property Name="Text" Value="Apply" />
62        <Property Name="UnifiedAreaRect" Value="
              {{0.583,0},{0.825,0},{0.768,0},{0.969,0}}" />
63      </Window>
64    </Window>
65 </GUILayout>
```

Para comenzar, la línea 2 define el tipo de *script*, como se ha explicado en la Sección anterior. El tipo de *Window* que contendrá al resto es un *TaharezLook/FrameWindow*, y se le ha puesto el nombre *Cfg*. Es importante que el tipo del *Window* indique el esquema al que pertenece, por eso se antepone *TaharezLook/*. A partir de aquí se añaden el resto de *Widgets*.

En total se han añadido 10 *Widgets* más a la ventana *Cfg*:

- Una etiqueta (*StaticText*) con el texto “Sound Volume” (llamada “Cfg/SndText” - línea 9) y un *Spinner* para indicar el valor (llamado “Cfg/SndVolume” - línea 13).
- Una etiqueta con el texto “Fullscreen” (línea 22) y un *Checkbox* para activarlo y desactivarlo (línea 26).
- Una etiqueta con el texto “Port”(línea 30) y un *EditBox* para indicar el número de puerto (línea 34).
- Una etiqueta con el texto “Resolution” (línea 41) y un *ItemListBox* para elegir entre varias opciones (45).
- Un botón (*Button*) con el texto “Exit” para terminar la aplicación (línea 55).
- Un botón con el texto “Apply” para aplicar los cambios (línea 60).

Cada uno de los *Window*s tiene unas propiedades para personalizarlos. En la documentación se explican todas y cada una de las opciones de los *Window* en función del esquema que se utilice.

Para indicar el valor de una propiedad en un fichero de *script* cualquiera de CEGUI, se indica en el campo *Value*, y siempre entre comillas, ya sea un número, una cadena o una palabra reservada.

Estas son algunas de las propiedades utilizadas:

- *Text:* indica el texto del *Widget*. Por ejemplo, la etiqueta de un botón o el título de una ventana.

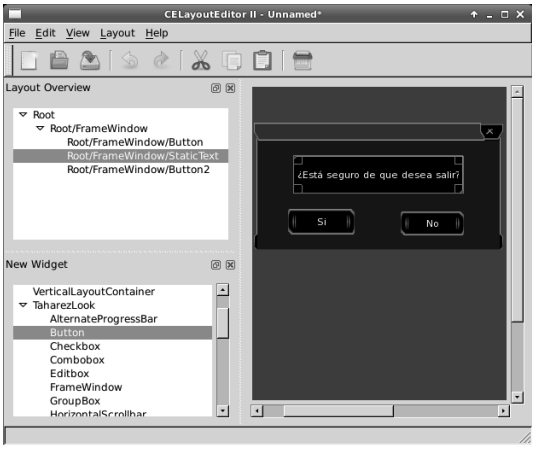


Figura 14.8: Interfaz de CEGUI Layout Editor II.

- *UnifiedAreaRect:* es uno de los más importantes. Indica la posición y el tamaño, mediante un objeto *URect*, de un *Window* **relativo a su padre**. Como se indicó en secciones anteriores, se trata de cuatro *UDim*s (cada uno de ellos con un factor de escala relativo y un offset), para indicar la esquina superior izquierda del rectángulo (los dos primeros *UDim*s), y la inferior derecha.

  Es importante tener en cuenta que si al *Widget* hijo se le indique que ocupe todo el espacio (con el valor para la propiedad de {{0,0},{0,0},{1,0},{1,0}}), ocupará todo el espacio del *Window* al que pertenezca, no necesariamente toda la pantalla.

- *TitlebarFont:* indica el tipo de fuente utilizado en el título de la barra de una *FrameWindow*.

- *TitlebarEnabled:* activa o desactiva la barra de título de una *FrameWindow*.

- *CurrentValue:* valor actual de un *Widget* al que haya que indicárselo, como el *Spinner*.

- *MaximumValue* y *MinimumValue*: acota el rango de valores que puede tomar un *Widget*.

Cada *Widget* tiene sus propiedades y uso especial. Por ejemplo, el *Widget* utilizado para escoger la resolución (un *ItemListBox*), contiene a su vez otros dos *Window* del tipo *ListBoxItem*, que representan cada una de las opciones (líneas 47 y 50).

Para poder añadir funcionalidad a estos *Widgets* (ya sea asociar una acción o utilizar valores, por ejemplo), se deben recuperar desde código mediante el *WindowManager*, con el mismo nombre que se ha especificado en el *layout*. Este *script* por tanto se utiliza únicamente para cambiar la organización y apariencia de la interfaz, pero no su funcionalidad.

En este ejemplo concreto, sólo se ha añadido una acción al botón con la etiqueta “Exit” para cerrar la aplicación.

14.7. Editores de *layouts* gráficos

Este método para diseñar interfaces de usuario directamente modificando los ficheros xml *.layout* puede ser muy costoso. Existen un par de aplicaciones gráficas para el diseño de las interfaces. La primera *CEGUI Layout Editor* es muy avanzada, pero se ha abandonado. Actualmente se está desarrollando un editor nuevo y actualizado, *CEGUI Unified Editor*, pero todavía no se considera estable.

Para compilar el editor, es necesario recompilar CEGUI con soporte para Python, pues es necesaria la biblioteca *PyCEGUI*. Para compilarla en Ubuntu 11.04 y 11.10 se puede seguir un tutorial disponible en la web de CEGUI[2], en el apartado *HowTo*, el tutorial "Build PyCEGUI from source for Linux".

Se puede encontrar un tutorial completo de cómo instalar el editor en:

www.cegui.org.uk/wiki/index.php/CEED

Al tratarse de una versión inestable, es mejor consultar este tutorial para mantener los pasos actualizados de acuerdo a la última versión en desarrollo.

14.8. Scripts en detalle

En esta Sección se va a ver la estructura del resto de *scripts* que se utilizan para construir la interfaz. Es importante conocerlos para poder cambiar la apariencia y poder adaptarlas a las necesidades artísticas del proyecto.

14.8.1. *Scheme*

Los esquemas contienen toda la información para ofrecer *Widgets* a una interfaz, su apariencia, su comportamiento visual o su tipo de letra.

Listado 14.11: Estructura de un *script scheme*.

```
1 <?xml version="1.0" ?>
2 <GUIScheme Name="TaharezLook">
3     <Imageset Filename="TaharezLook.imageset" />
4     <Font Filename="DejaVuSans-10.font" />
5     <LookNFeel Filename="TaharezLook.looknfeel" />
6     <WindowRendererSet Filename="CEGUIFalagardWRBase" />
7     <FalagardMapping WindowType="TaharezLook/Button"
          TargetType="CEGUI/PushButton"  Renderer="Falagard/Button"
                LookNFeel="TaharezLook/Button" />
8     <FalagardMapping WindowType="TaharezLook/Checkbox"
          TargetType="CEGUI/Checkbox"    Renderer="Falagard/
          ToggleButton" LookNFeel="TaharezLook/Checkbox" />
9 </GUIScheme>
```

Al igual que sucedía con los *layout*, comienzan con una etiqueta que identifican el tipo de *script*. Esta etiqueta es *GUIScheme*, en la línea 2.

De las líneas 3-6 se indican los *scripts* que utilizará el esquema de los otros tipos. Qué conjunto de imágenes para los botones, cursores y otro tipo de *Widgets* mediante el *Imageset* (línea 3), qué fuentes utilizará mediante un *Font* (línea 4), y el comportamiento visual de los *Widgets* a través del *LookNFeel* (línea 5). Además se indica qué sistema de renderizado de *skin* utilizará (línea 6). CEGUI utiliza *Falagard*.

El resto se dedica a declarar el conjunto de *Widgets*. Para ello realiza un mapeado entre el *Widget* que habrá disponible en el esquema (por ejemplo, "TaharezLook/Button", en la línea 7), y los que ofrece *CEGUI*. Además, por cada uno se indican varios parámetros.

De este *script* usualmente se suele cambiar los *Imageset* y los *Font* para cambiar la apariencia de la interfaz, y suministrar los desarrollados de forma propia.

14.8.2. *Font*

Describen los tipos de fuente que se utilizarán en la interfaz.

Listado 14.12: Estructura de un *script font*.

```
1 <?xml version="1.0" ?>
2 <Font Name="DejaVuSans-10" Filename="DejaVuSans.ttf" Type="FreeType
      " Size="10" NativeHorzRes="800" NativeVertRes="600" AutoScaled=
      "true"/>
```

En este ejemplo, sólo se define un tipo de fuente, en la línea 2. Además de asignarle un nombre para poder usarla con CEGUI, se indica cuál es el fichero *ttf* que contiene la fuente, y otras propiedades como el tamaño o la resolución. Dentro de este fichero se puede añadir más de una fuente, añadiendo más etiquetas del tipo *Font*.

14.8.3. *Imageset*

Contiene las verdaderas imágenes que compondrán la interfaz. Este recurso es, junto al de las fuentes, los que más sujetos a cambio están para poder adaptar la apariencia de la interfaz a la del videojuego.

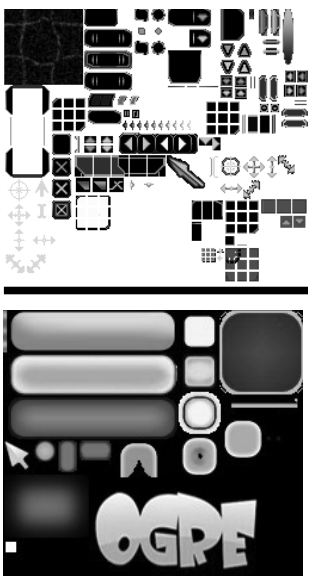


Figura 14.9: Matriz de imágenes que componen la interfaz del esquema *TaharezLook* (arriba), y *OgreTray* (abajo).

Listado 14.13: Estructura de un *script imageset*.

```
1 <?xml version="1.0" ?>
2 <Imageset Name="TaharezLook" Imagefile="TaharezLook.tga"
      NativeHorzRes="800" NativeVertRes="600" AutoScaled="true">
3     <Image Name="MouseArrow" XPos="138" YPos="127" Width="31"
          Height="25" XOffset="0" YOffset="0" />
4 </Imageset>
```

CEGUI almacena las imágenes que componen la interfaz como un conjunto de imágenes. De este modo, CEGUI sólo trabaja con un archivo de imagen, y dentro de ella debe saber qué porción corresponde a cada elemento. En la Figura 14.9 podemos ver el archivo de imagen que contiene toda la apariencia del esquema *TaharezLook* y *OgreTray*.

En la línea 2 del *script*, se indica el nombre del conjunto de imágenes y cuál es el archivo de imagen que las contendrá, en este caso, *TaharezLook.tga*. A partir de ese archivo, se define cada uno de los elementos indicando en qué región de *TaharezLook.tga* se encuentra. En la línea 3 se indica que el cursor del ratón ("MouseArrow") se encuentra en el rectángulo definido por la esquina superior izquierda en la posición (138, 127), y con unas dimensiones de 31x25.

De esta forma, se puede diseñar toda la interfaz con un programa de dibujo, unirlos todos en un archivo, e indicar en el *script imageset* dónde se encuentra cada una. Para ver a qué más elementos se les puede añadir una imagen, consultar la referencia, o estudiar los *imageset* proporcionados por CEGUI.

14.8.4. *LookNFeel*

Estos tipos de *scripts* son mucho más complejos, y los que CEGUI proporciona suelen ser más que suficiente para cualquier interfaz. Aún así, en la documentación se puede consultar su estructura y contenido.

14.9. Cámara de Ogre en un *Window*

Una característica muy interesante que se puede realizar con CEGUI es mostrar lo que capta una cámara de Ogre en un *Widget*. Puede ser interesante para mostrar mapas o la vista trasera de un coche de carreras, por ejemplo.

Como se aprecia en la Figura 14.10, la aplicación mostrará a Sinbad de la forma habitual, pero además habrá una ventana que muestra otra vista distinta, y ésta a su vez contendrá un botón para salir de la aplicación.

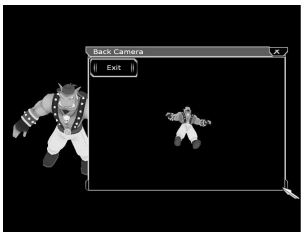


Figura 14.10: *Screenshot* de la aplicación de ejemplo.

A continuación se muestra el fichero del *layout*. Después de haber visto en la Sección 14.5 cómo funcionan los *layouts*, este no tiene ninguna complicación. Se trata de una única *FrameWindow* (línea 3) llamada "CamWin", con el título "Back Camera" y una posición y tamaño determinados. Esta a su vez contiene dos *Window*s más: uno del tipo *StaticImage* (línea 8), que servirá para mostrar la imagen de la textura generada a partir de una cámara secundaria de Ogre, para tener ese segundo punto de vista; y uno del tipo *Button* (línea 11), con el texto "Exit".

El único aspecto resaltable de este código es que el *Widget* que muestra la imagen ("CamWin/RTTWindow") ocupa todo el area de su padre (su propiedad *UnifiedAreaRect* vale {{0,0},{0,0,},{1,0},{1,0}}. Esto quiere decir que ocupará toda la ventana "CamWin", que no toda la pantalla.

Listado 14.14: *Layout* **del ejemplo.**

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <GUILayout>
3    <Window Type="TaharezLook/FrameWindow" Name="CamWin" >
4      <Property Name="Text" Value="Back Camera" />
5      <Property Name="TitlebarFont" Value="DejaVuSans-10" />
6      <Property Name="TitlebarEnabled" Value="True" />
7      <Property Name="UnifiedAreaRect" Value="
           {{0.6,0},{0.6,0},{0.99,0},{0.99,0}}" />
8      <Window Type="TaharezLook/StaticImage" Name="CamWin/RTTWindow"
           >
9        <Property Name="UnifiedAreaRect" Value="
             {{0,0},{0,0},{1,0},{1,0}}" />
10     </Window>
11     <Window Type="TaharezLook/Button" Name="ExitButton" >
12       <Property Name="Text" Value="Exit" />
13       <Property Name="UnifiedAreaRect" Value="
             {{0.01,0},{0.01,0},{0.25,0},{0.15,0}}" />
14     </Window>
15   </Window>
16 </GUILayout>
```

Una vez tenemos la organización de la interfaz, es necesario dotarla de funcionalidad. A continuación se muestran las modificaciones que hay que añadir para que se pueda renderizar una cámara de Ogre en el *Widget*.

Listado 14.15: Inicialización de la textura y del *Widget* que la mostrará.

```
1  Ogre::Camera* _camBack = _sceneManager->createCamera("BackCamera");
2  _camBack->setPosition(Ogre::Vector3(-5,-20,20));
3  _camBack->lookAt(Ogre::Vector3(0,0,0));
4  _camBack->setNearClipDistance(5);
5  _camBack->setFarClipDistance(10000);
6  _camBack->setAspectRatio(width / height);
7
8  Ogre::TexturePtr tex = _root->getTextureManager()->createManual(
9                "RTT",
10               Ogre::ResourceGroupManager::
                     DEFAULT_RESOURCE_GROUP_NAME,
11               Ogre::TEX_TYPE_2D,
12               512,
13               512,
14               0,
15               Ogre::PF_R8G8B8,
16               Ogre::TU_RENDERTARGET);
17
18 Ogre::RenderTexture* rtex = tex->getBuffer()->getRenderTarget();
19
20 Ogre::Viewport* v = rtex->addViewport(_camBack);
21 v->setOverlaysEnabled(false);
22 v->setClearEveryFrame(true);
23 v->setBackgroundColour(Ogre::ColourValue::Black);
24
25 CEGUI::Texture& guiTex = renderer->createTexture(tex);
26
27 CEGUI::Imageset& imageSet = CEGUI::ImagesetManager::getSingleton().
       create("RTTImageset", guiTex);
28 imageSet.defineImage("RTTImage",
29              CEGUI::Point(0.0f,0.0f),
30              CEGUI::Size(guiTex.getSize().d_width, guiTex.getSize()
                    .d_height),
31              CEGUI::Point(0.0f,0.0f));
32
33 CEGUI::Window* ex1 = CEGUI::WindowManager::getSingleton().
       loadWindowLayout("render.layout");
34
35 CEGUI::Window* RTTWindow = CEGUI::WindowManager::getSingleton().
       getWindow("CamWin/RTTWindow");
36
37 RTTWindow->setProperty("Image",CEGUI::PropertyHelper::imageToString
       (&imageSet.getImage("RTTImage")));
38
39 //Exit button
40 CEGUI::Window* exitButton = CEGUI::WindowManager::getSingleton().
       getWindow("ExitButton");
41 exitButton->subscribeEvent(CEGUI::PushButton::EventClicked,
42                CEGUI::Event::Subscriber(&MyFrameListener::quit,
43                                _framelistener));
44 //Attaching layout
45 sheet->addChildWindow(ex1);
46 CEGUI::System::getSingleton().setGUISheet(sheet);
```

Esta parte supone más código de Ogre que de CEGUI.

Puesto que el objetivo es mostrar en un *Widget* lo que está capturando una cámara, el primer paso es crearla. De las líneas 1 a las 6 se crea una *Ogre::Camera* de forma convencional. Se indica la posición, hacia dónde mira, los planos de corte *Near* y *Far*, y el *aspect ratio*.

Por otro lado, hay que crear la textura en la que se volcará la imagen de la cámara. Para ello se utiliza la función `createManual()` de *TexureManager*, en las líneas 8 a la 16. La textura se llama "RTT", tendrá un tamaño de 512x512, y será del tipo *Ogre::TU_RENDERTARGET*, para que pueda albergar la imagen de una cámara.

El siguiente paso es crear el *ViewPort*. En la línea 18 se obtiene el objeto *RenderTexture* a partir de la textura manual creada. En la línea 20 se obtiene el objeto *ViewPort* a partir del *RenderTexture* y utilizando la cámara que se quiere mostrar. En este caso, *_camBack*. A este *ViewPort* se le indica que no dibuje los *Overlays* (línea 21), aunque podría hacerlo sin problemas, y que se actualice en cada frame (línea 22). Además se establece el color de fondo en negro (línea 23).

Hasta ahora se ha creado la textura *tex* de Ogre que es capaz de actualizarse con la imagen capturada por una cámara, también de Ogre, para tal efecto. El siguiente paso es preparar CEGUI para mostrar imagen en uno de sus *Widgets*, y que además esa imagen la obtenga de la textura de Ogre.

En la línea 25 se crea la textura *guiTex* de CEGUI. El objeto *renderer* específico para Ogre proporciona la función `createTexture()`, que la crea a partir de una de Ogre.

Como se vio en la Sección 14.8.3, CEGUI está diseñado para tratar las distintas imágenes como porciones de un array de imágenes (ver Figura 14.9).

De este modo, primeramente hay que crear un *Imageset* a partir de la textura que devuelve Ogre (ya en formato de CEGUI) en la línea 27. A este conjunto de imágenes se le ha llamado "RTTImageset". Después, hay que identificar qué porción corresponde a la textura de la cámara de ese *Imageset*. En este caso, es la textura completa, por lo que en la línea 28 se define la imagen con el nombre "RTTImage". El primer parámetro es el nombre, el segundo la esquina superior izquierda de la porción que define la imagen (se indica el 0,0), el tercero el tamaño de la porción, que corresponde al tamaño de la textura, y el cuarto un *offset*.

Ya se ha conseguido obtener una imagen de CEGUI que se actualizará cada frame con lo que capturará la cámara. Lo único que falta es recuperar los *Window* definidos en el *layout* e indicar que el *Widget* "CamWin/RTTWindow" muestre dicha textura.

En la línea 33 se carga el *layout* como se hizo en anteriores ejemplos. Es importante hacerlo antes de comenzar a recuperar los *Window*s definidos en el *layout*, porque de lo contrario no los encontrará.

Se recupera el *Window* que mostrará la imagen (del tipo *StaticImage*), llamado "CamWin/RTTWindow", en la línea 35. En la siguiente línea, la 37, se indica en la propiedad "Image" de dicho *Window* que utilice la imagen "RTTImage" del conjunto de imagenes.

Con esto ya es suficiente para mostrar en el *Widget* la imagen de la cámara. Por último se añade la funcionalidad al botón de salida, en las líneas 40 y 41, como en anteriores ejemplos, y se añade el *layout* al *Sheet* (línea 45) y se establece dicho *Sheet* por defecto (línea 46).

14.10. Formateo de texto

Una característica muy versátil que ofrece CEGUI es la de proporcionar formato a las cadenas de texto mediante el uso de *tags* (etiquetas). Este formato permite cambiar el color, tamaño, alineación o incluso insertar imágenes. A continuación se describe el funcionamiento de estas etiquetas.

14.10.1. Introducción

El formato de las etiquetas utilizadas son el siguiente:

```
[tag-name='value']
```

Estas etiquetas se insertan directamente en las cadenas de texto, y funcionan como estados. Es decir, si se activa una etiqueta con un determinado color, se aplicará a todo el texto que le preceda a no ser que se cambie explícitamente con otra etiqueta.

A continuación se muestran los diferentes aspectos que se pueden personalizar con esta técnica. Al final se muestra una aplicación de ejemplo que implementa su funcionamiento.

Si se quiere mostrar como texto la cadena "[Texto]" sin que lo interprete como una etiqueta, es necesario utilizar un carácter de escape. En el caso concreto de C++ se debe anteponer "\\" únicamente a la primera llave, de la forma "\\[Texto]"

14.10.2. Color

Para cambiar el color del texto, se utiliza la etiqueta "colour", usada de la siguiente forma:

```
[colour='FFFF0000']
```

Esta etiqueta colorea el texto que la siguiese de color rojo. El formato en el que se expresa el color es *ARGB* de 8 bits, es decir '*AARRGGBB*'. El primer parámetro expresa la componente de transparencia alpha, y el resto la componente RGB.

14.10.3. Formato

Para cambiar el formato de la fuente (tipo de letra, negrita o cursiva, por ejemplo) es más complejo ya que se necesitan los propios ficheros que definan ese tipo de fuente, y además deben estar definidos en el *script Font*.

Estando seguro de tener los archivos *.font* y de tenerlos incluidos dentro del fichero del esquema, se puede cambiar el formato utilizando la etiqueta:

```
[font='Arial-Bold-10']
```

Esta en concreto corresponde al formato en negrita del tipo de letra Arial, de tamaño 10. El nombre que se le indica a la etiqueta es el que se especificó en el *.scheme*.

14.10.4. Insertar imágenes

Insertar una imagen dentro de una cadena, al estilo de los emoticonos, es sencillo. La estructura de la etiqueta es la siguiente:

```
[imageset='set:<imageset> image:<image>']
```

Una vez más, CEGUI trata las imágenes individuales como parte de un *Imageset*, por lo que hay que indicarle el conjunto, y la imagen.

Aprovechando el *Imageset* que utiliza la interfaz, "TaharezLook", se va a mostrar una de ellas, por ejemplo la equis para cerrar la ventana. Echando un vistazo al *xml*, se puede identificar que la imagen que se corresponde con la equis se llama "CloseButtonNormal". La etiqueta que habría que utilizar sería la siguiente:

```
[image='set:TaharezLook image=CloseButtonNormal']
```

Además, existe otra etiqueta poder cambiar el tamaño de las imágenes insertadas. El formato de la etiqueta es el siguiente:

```
[image-size='w:<width_value> h:<height_value>']
```

El valor del ancho y del alto se da en píxeles, y debe ponerse antes de la etiqueta que inserta la imagen. Para mostrar las imágenes en su tamaño original, se deben poner los valores de *width* y *height* a cero.

14.10.5. Alineamiento vertical

Cuando un texto contiene distintos tamaños de letra, es posible configurar el alineamiento vertical de cada parte. El alto de una línea concreta vendrá definido por el alto del texto con mayor tamaño. El resto de texto, con menor tamaño, podrá alinearse verticalmente dentro de ese espacio.

Los tipos de alineamiento vertical disponibles son:

- *top*: lo alinea hacia arriba.
- *bottom*: lo alinea hacia abajo.
- *center*: lo centra verticalmente.
- *strecth*: lo *estira* verticalmente para ocupar todo el alto.

El formato de la etiqueta es:

```
[vert-alignment='<tipo_de_alineamiento>']
```

14.10.6. *Padding*

El *padding* consiste en reservar un espacio alrededor del texto que se desee. Para definirlo, se indican los píxeles para el *padding* izquierdo, derecho, superior e inferior. Así, además de el espacio que ocupe una determinada cadena, se reservará *como un margen* el espacio indicado en el *padding*. Viendo la aplicación de ejemplo se puede apreciar mejor este concepto.

El formato de la etiqueta es:

```
[padding='l:<left_padding> t:<top_padding> r:<right_padding> b:<
    bottom_padding>']
```

Para eliminar el *padding*, utilizar la etiqueta con los valores a 0.

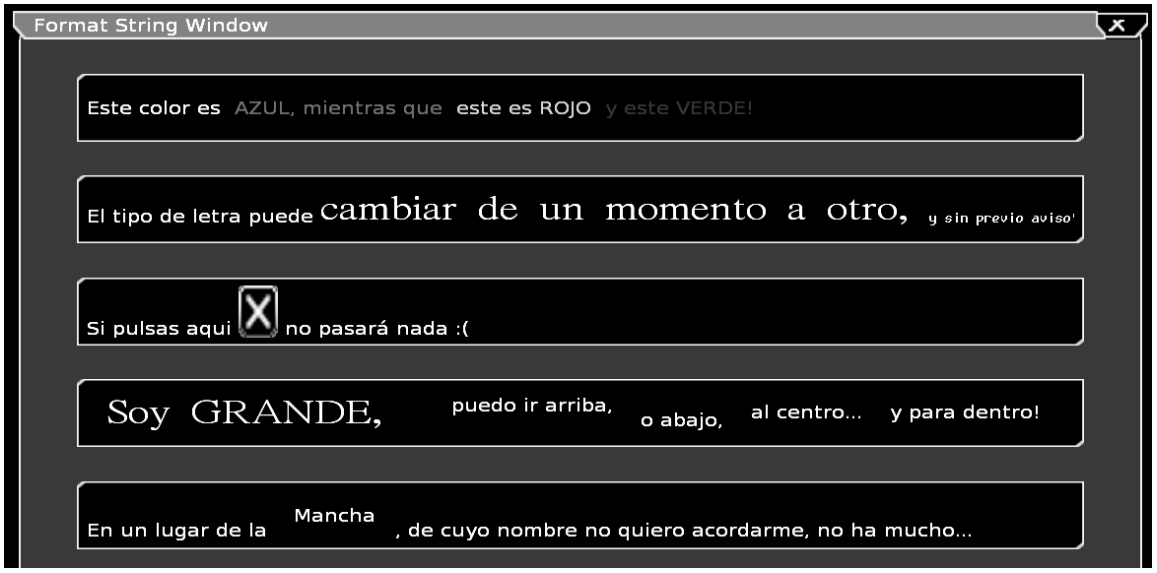


Figura 14.11: Ejemplos de uso del formateo de cadenas.

14.10.7. Ejemplo de texto formateado

El siguiente es un ejemplo que muestra algunas de las características que se han descrito. En la Figura 14.11 se aprecia el acabado final. El siguiente es el *layout* utilizado:

Listado 14.16: *Layout* de la aplicación.

```
<?xml version="1.0" encoding="UTF-8"?>
<GUILayout>
  <Window Type="TaharezLook/FrameWindow" Name="FormatWin">
    <Property Name="Text" Value="Format String Window"/>
    <Property Name="TitlebarEnabled" Value="True"/>
    <Property Name="UnifiedAreaRect" Value="
        {{0.05,0},{0.05,0},{0.95,0},{0.95,0}}"/>
    <!-- Static Text -->
    <Window Type="TaharezLook/StaticText" Name="FormatWin/Text1">
      <Property Name="UnifiedAreaRect" Value="
          {{0.05,0},{0.05,0},{0.95,0},{0.15,0}}"/>
    </Window>
    <!-- Other Static Text ... -->
    <!-- Exit Button -->
    <Window Type="TaharezLook/Button" Name="FormatWin/ExitButton">
      <Property Name="Text" Value="Exit" />
      <Property Name="UnifiedAreaRect" Value="
          {{0,0},{0.95,0},{1,0},{1,0}}"/>
    </Window>
  </Window>
</GUILayout>
```

Y el siguiente listado muestra cada una de las cadenas que se han utilizado, junto a las etiquetas:

Listado 14.17: Código con los *tags* de formateo.

```
"Este color es [colour='FFFF0000'] AZUL, mientras que [colour='
    FF00FF00'] este es ROJO [colour='FF0000FF'] y este VERDE!"
```

```
2
3 "El tipo de letra puede [font='Batang-26']cambiar de un momento a
      otro, [font='fkp-16']y sin previo aviso!"
4
5 "Si pulsas aqui [image-size='w:40 h:55'][image='set:TaharezLook
      image:CloseButtonNormal'] no pasara nada :("
6
7 "[font='Batang-26'] Soy GRANDE,   [font='DejaVuSans-10'][vert-
      alignment='top'] puedo ir arriba,    [vert-alignment='bottom']o
       abajo,    [vert-alignment='centre']al centro..."
8
9 "En un lugar de la [padding='l:20 t:15 r:20 b:15']Mancha[padding='l
      :0 t:0 r:0 b:0'], de cuyo nombre no quiero acordarme, no ha
      mucho..."
```

La primera cadena (línea 1) utiliza las etiquetas del tipo *colour* para cambiar el color del texto escrito a partir de ella. Se utilizan los colores rojo, verde y azul, en ese orden.

La segunda (línea 3) muestra cómo se pueden utilizar las etiquetas para cambiar totalmente el tipo de fuente, siempre y cuando estén definidos los recursos *.font* y estos estén reflejados dentro el *.scheme*.

La tercera (línea 5) muestra una imagen insertada en medio del texto, y además redimensionada. Para ello se utiliza la etiqueta de redimensionado para cambiar el tamaño a 40x55, y después inserta la imagen "CloseButtonNormal", del conjunto "TaharezLook"

La cuarta (línea 7) muestra una cadena con un texto ("Soy GRANDE") de un tipo de fuente con un tamaño 30, y el resto con un tamaño 10. Para el resto del texto, sobra espacio vertical, por lo que se utiliza la etiqueta *vertical-alignment* para indicar dónde posicionarlo.

Por último, la quinta cadena (línea 9), utiliza *padding* para la palabra "Mancha". A esta palabra se le reserva un *margen* izquierdo y derecho de 20 píxeles, y un superior e inferior de 15.

14.11. Características avanzadas

CEGUI es una biblioteca muy potente y flexible que puede ser utilizada junto a muchas otras para crear efectos visualmente muy impactantes. Algunas características avanzadas que se han implementado son efectos de ventanas, como transparencia y aspecto gelatinoso, o incluso incrustar un navegador dentro de una ventana.

Para aprender estas características y más, en su página existen muchos manuales, y dispone de una muy buena documentación [2].

Capítulo 15

Materiales y Texturas

Carlos González Morcillo

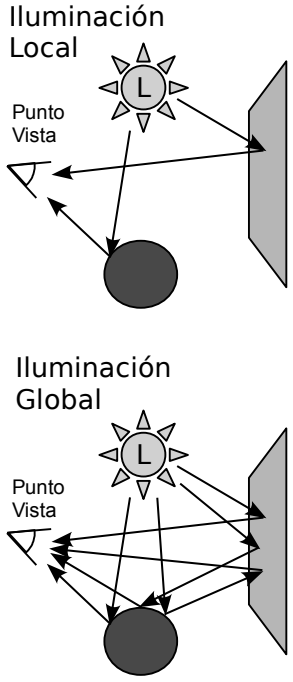


Figura 15.1: Diferencias entre los modelos de iluminación local y global.

En este capítulo estudiaremos los conceptos fundamentales con la definición de materiales y texturas. Introduciremos la relación entre los modos de sombreado y la interacción con las fuentes de luz, describiendo los modelos básicos soportados en aplicaciones interactivas. Para finalizar, estudiaremos la potente aproximación de Ogre para la definición de materiales, basándose en los conceptos de técnicas y pasadas.

15.1. Introducción

Los materiales describen las propiedades físicas de los objetos relativas a cómo reflejan la luz incidente. Obviamente, el aspecto final obtenido será dependiente tanto de las propiedades del material, como de la propia definición de las fuentes de luz. De este modo, materiales e iluminación están íntimamente relacionados.

Desde los inicios del estudio de la óptica, investigadores del campo de la física han desarrollado modelos matemáticos para estudiar la interacción de la luz en las superficies. Con la aparición del microprocesador, los ordenadores tuvieron suficiente potencia como para poder simular estas complejas interacciones.

Así, usando un ordenador y partiendo de las propiedades geométricas y de materiales especificadas numéricamente es posible simular la reflexión y propagación de la luz en una escena. A mayor precisión, mayor nivel de realismo en la imagen resultado.

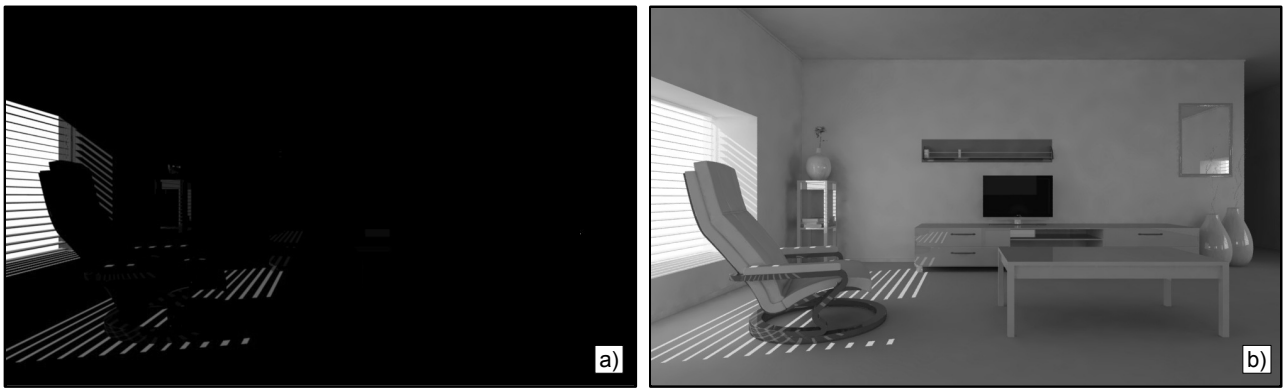


Figura 15.2: Ejemplo de resultado utilizando un modelo de iluminación local y un modelo de iluminación global con la misma configuración de fuentes de luz y propiedades de materiales. **a)** En el modelo de iluminación local, si no existen fuentes de luz en el interior de la habitación, los objetos aparecerán totalmente "*a oscuras*", ya que no se calculan los rebotes de luz indirectos. **b)** Los modelos de iluminación global tienen en cuenta esas contribuciones relativas a los rebotes de luz indirectos.

Esta conexión entre la simulación del comportamiento de la luz y el nivel de realismo queda patente en las aproximaciones existentes de diferentes métodos de render. Una ecuación que modela el comportamiento físico de la luz, ampliamente aceptada por la comunidad, es la propuesta por *Kajiya* en 1986. De forma general podemos decir que a mayor simplificación en la resolución de los términos de esta ecuación tendremos métodos menos realistas (y computacionalmente menos costosos).

A un alto nivel de abstracción, podemos realizar una primera taxonomía de métodos de render entre aquellos que realizan una simulación de **iluminación local**, teniendo en cuenta únicamente una interacción de la luz con las superficies, o los métodos de **iluminación global** que tratan de calcular *todas* [1] las interacciones de la luz con las superficies de la escena. La Figura 15.2 muestra el resultado de renderizar la misma escena con un método de iluminación local y uno global. Los modelos de iluminación global incorporan la iluminación directa que proviene de la primera interacción de las superficies con las fuentes de luz, así como la iluminación indirecta reflejada por otras superficies existentes en la escena.

La potencia de cálculo actual hace inviable el uso de métodos de iluminación global. Se emplean aproximaciones de precálculo de la iluminación, que serán estudiadas en el capítulo de iluminación. Así, en los motores gráficos actuales como Ogre, los materiales definen cómo se refleja la luz en los objetos (pero no su contribución con otros objetos), empleando un esquema de **iluminación local**.

15.2. Modelos de Sombreado

Como hemos comentado anteriormente, es habitual en gráficos por computador interactivos (y en el caso de videojuegos especialmente) emplear modelos de iluminación local. En cierto modo, el sombrado es equivalente a *pintar con luz*. En este apartado estudiaremos los modelos de sombreado más ampliamente utilizados en gráficos interactivos, que fueron inicialmente desarrollados en los años 70.

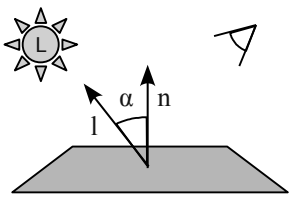


Figura 15.3: Modelo de sombreado difuso básico de Lambert en el que el color c se define como $c \propto n \cdot l$. Los vectores n y l deben estar normalizados.

[1] Debido a que es imposible calcular las infinitas interacciones de los rayos de luz con todos los objetos de la escena, las aproximaciones de iluminación global se ocuparán de calcular algunas de estas interacciones, tratando de minimizar el error de muestreo.

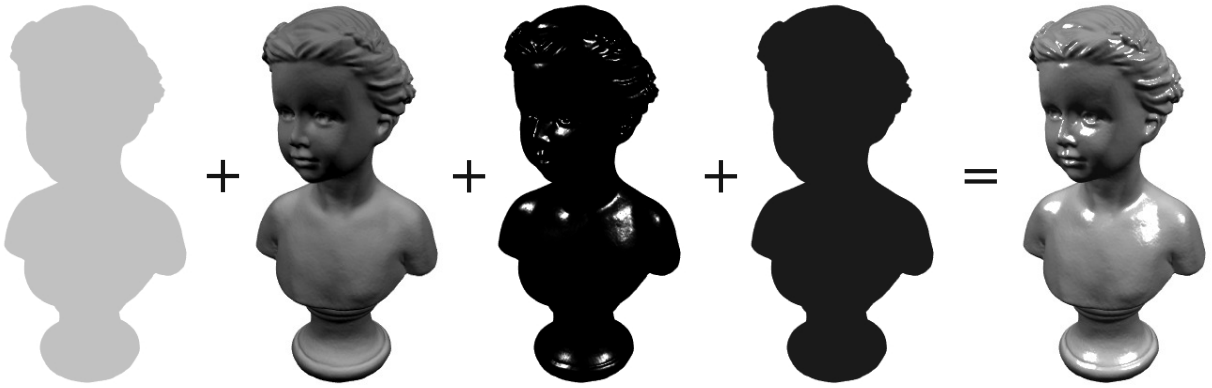


Figura 15.4: Modos básicos de sombreado de iluminación local.

- **Sombreado difuso**. Muchos objetos del mundo real tienen una acabado eminentemente mate (sin brillo). Por ejemplo, el papel o una tiza pueden ser superficies con sombreado principalmente difuso. Este tipo de materiales reflejan la luz en todas las direcciones, debido principalmente a las rugosidades microscópicas del material. Como efecto visual, el resultado de la iluminación es mayor cuando la luz incide perpendicularmente en la superficie. La intensidad final viene determinada por el ángulo que forma la luz y la superficie (es independiente del punto de vista del observador). En su expresión más simple, el modelo de reflexión difuso de *Lambert* dice que el color de una superficie es proporcional al coseno del ángulo formado entre la normal de la superficie y el vector de dirección de la fuente de luz (ver Figura 15.3).

- **Sombreado especular**. Es el empleado para simular los *brillos* de algunas superficies, como materiales pulidos, pintura plástica, etc. Una característica principal del sombreado especular es que el brillo se *mueve con el observador*. Esto implica que es necesario tener en cuenta el vector del observador. La *dureza* del brillo (la cantidad de reflejo del mismo) viene determinada por un parámetro *h*. A mayor valor del parámetro *h*, más concentrado será el brillo. El comportamiento de este modelo de sombreado está representado en la Figura 15.5, donde r es el vector reflejado del l (forma el mismo ángulo α con n) y e es el vector que se dirige del punto de sombreado al observador. De esta forma, el color final de la superficie es proporcional al ángulo θ.

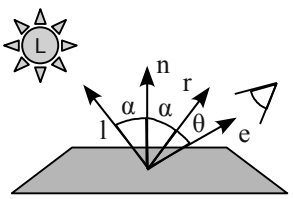


Figura 15.5: En el modelo de sombreado especular, el color c se obtiene como $c \propto (r \cdot e)^h$. Los vectores r y e deben estar normalizados.

- **Sombreado ambiental**. Esta componente básica permite añadir una *aproximación* a la *iluminación global* de la escena. Simplemente añade un color base independiente de la posición del observador y de la fuente de luz. De esta forma se evitan los tonos *absolutamente negros* debidos a la falta de iluminación global, añadiendo este término constante. En la Figura 15.4 se puede ver la componente ambiental de un objeto sencillo.

- **Sombreado de emisión**. Finalmente este término permite añadir una simulación de la iluminación propia del objeto. No obstante, debido a la falta de interacción con otras superficies (incluso con las caras poligonales del propio objeto), suele emplearse como una alternativa al sombreado ambiental a nivel local. El efecto es como tener un objeto que emite luz pero cuyos rayos no interactúan con ninguna superficie de la escena (ver Figura 15.4).

El sombreado final de la superficie se obtiene como combinación de los cuatro modos de sombreado anteriores (ver Figura 15.4). Esta aproximación es una simplificación del modelo de reflexión físicamente correcto definido por la Función de Distribución de Reflactancia Bidireccional BRDF (Bidirectional Reflactance Distribution Function). Esta función define cómo se refleja la luz en cualquier superficie opaca, y es empleada en motores de *rendering* fotorrealistas.

15.3. Mapeado de Texturas

Los materiales definen propiedades que son constantes a lo largo de la superficie. Hasta ahora, hemos hablado de materiales básicos en las superficies, con propiedades (como el color) constantes.

Las texturas permiten variar estas propiedades, determinando en cada punto cómo cambian concretamente estas propiedades. Básicamente podemos distinguir dos tipos de texturas:

- **Texturas Procedurales.** Su valor se determina mediante una ecuación. Estas texturas se calculan rápidamente y no tienen requisitos de espacio en disco, por lo que son ampliamente utilizadas en síntesis de imagen realista para simular ciertos patrones existentes en la naturaleza (madera, mármol, nubes, etc).

  La Figura 15.6 muestra un ejemplo de este tipo de texturas. En videojuegos sin embargo, se emplean en menor medida, ya que resulta habitualmente más interesante emplear texturas de imagen con una resolución controlada.

- **Texturas de Imagen.** Almacenan los valores en una imagen, típicamente bidimensional.

```
color func(p3d p){
 if (sin(p_z)>0)
   return C_1
 else
   return C_2
}
```

Figura 15.6: Definición de una sencilla textura procedural que define bandas de color dependiendo del valor de coordenada Z del punto 3D.

Las texturas procedurales obtienen valores habitualmente en el espacio 3D, por lo que no es necesaria ninguna función de proyección de estas texturas sobre el objeto. Sin embargo, para utilizar las texturas de imagen es necesario indicar cómo queremos aplicar esa textura (2D) a la geometría del modelo 3D. Es decir, debemos espeicifcar cómo se recubrirá el objeto con ese mapa de textura. Existen varias alternativas para realizar este mapeado. Empleando proyecciones ortogonales es posible describir esta correspondencia. Se emplean cuatro modos básicos de proyección (ver Figura 15.8). Estos modos de proyección utilizas las coordenadas del objeto 3D normalizadas en el interior de una caja unitaria.

Una de las características interesantes de los modelos de mapeado de texturas (tanto ortogonales como mediante mapas paramétricos UV) es la **independencia de la resolución** de la imagen. De este modo es posible tener texturas de diferentes tamaños y emplearlas aplicando técnicas de nivel de detalle (*LOD*). Así, si un objeto se muestra a una gran distancia de la cámara es posible cargar texturas de menor resolución que requieran menor cantidad de memoria de la GPU.

Como hemos visto en capítulos anteriores, un método de proyección de texturas de imagen muy empleado en videojuegos es el mapeado paramétrico, también denominado mapeado UV. En este método se definen dos coordenadas paramétricas (entre 0 y 1) para cada vértice de cada cara del modelo (ver Figura 15.7). Estas coordenadas son independientes de la resolución de la imagen, por lo que permite cambiar en tiempo de

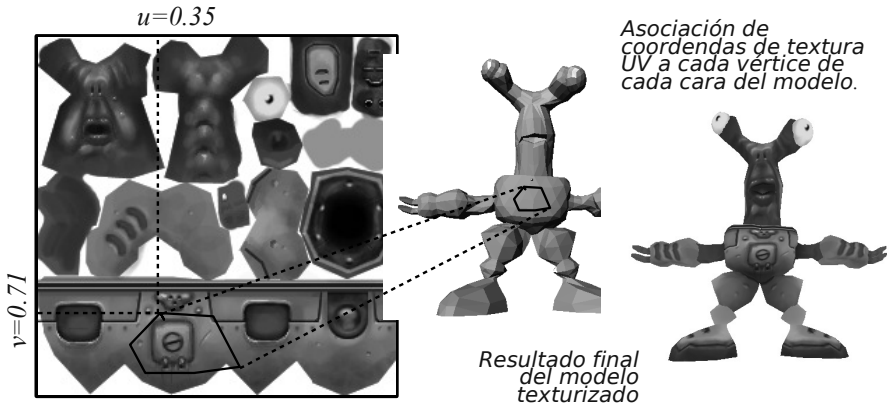


Figura 15.7: Asignación de coordenadas UV a un modelo poligonal. Esta operación suele realizarse con el soporte de alguna herramienta de edición 3D.

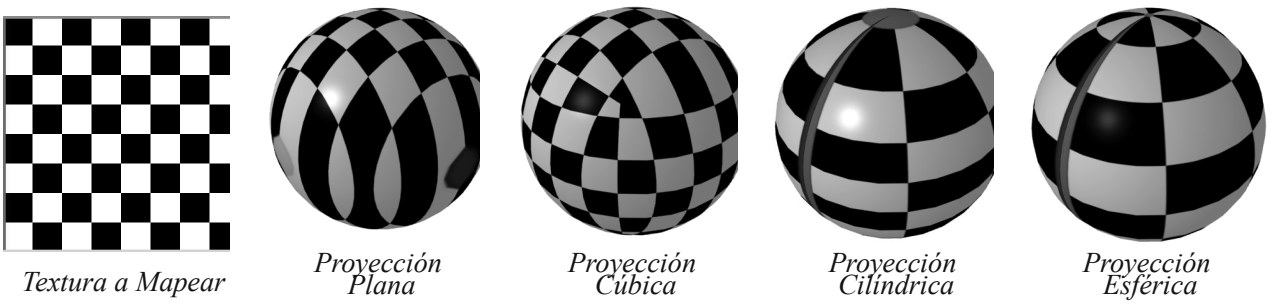


Figura 15.8: Métodos básicos de mapeado ortogonal sobre una esfera. Los bordes marcados con líneas de colores en la textura a mapear en la izquierda se proyectan de forma distinta empleado diversos métodos de proyección.

ejecución la textura teniendo en cuenta ciertos factores de distancia, importancia del objeto, etc. El mapeado UV permite pegar la textura al modelo de una forma muy precisa. Incluso si se aplica sobre el modelo deformación de vértices (*vertex blending*), el mapa seguirá aplicándose correctamente. Por esta razón y su alta eficiencia es una técnica ampliamente utilizada en gráficos por computador.

Shading programable

En esta sección estudiaremos únicamente lo que se conoce como *shading* fijo (*fixed shading*). En capítulos posteriores del documento estudiaremos cómo aplicar shaders programando la GPU (en *pixel shading* o *fragment shading*).

15.4. Materiales en Ogre

El despliegue de entidades en Ogre se realiza en paquetes, de modo que existe una relación directa entre el número de materiales y el número de *paquetes* que Ogre enviará a la tarjeta gráfica. Por ejemplo, si 10 elementos comparten el mismo material, podrán ser enviados en un único paquete a la GPU (en lugar de en 10 paquetes por separado), de modo que podrán compartir el mismo *estado interno*.

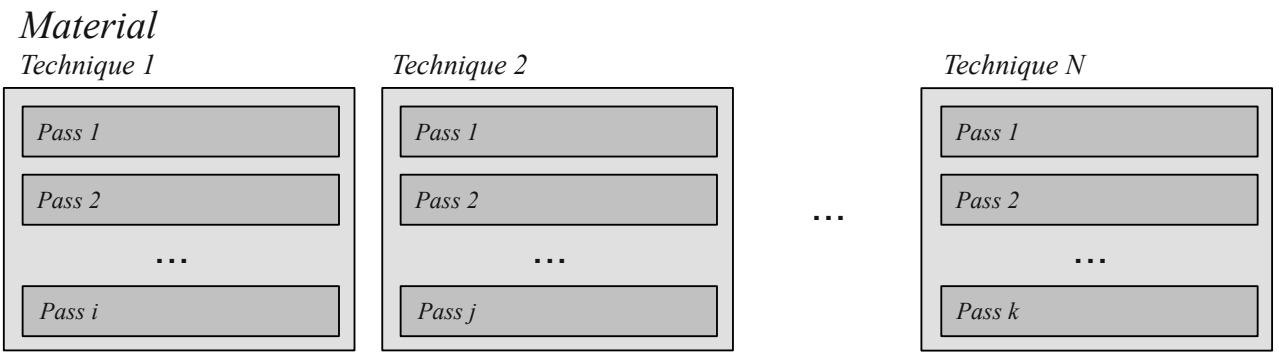


Figura 15.10: Descripción de un material en base a técnicas y pasadas. El número de técnicas y pasadas asociadas a cada técnica puede ser diferente.

Con la idea de realizar estas optimizaciones, Ogre define que por defecto los materiales son compartidos entre objetos. Esto implica que el mismo puntero que referencia a un material es compartido por todos los objetos que utilizan ese material. De este modo, si queremos cambiar la propiedad de un material de modo que únicamente afecte a un objeto es necesario **clonar** este material para que los cambios no se propaguen al resto de objetos.

Los materiales de Ogre se definen empleando técnicas y esquemas. Una **Técnica** puede definirse como cada uno de los modos *alternativos* en los que puede renderizarse un material. De este modo, es posible tener, por ejemplo, diferentes niveles de detalle asociados a un material. Los **esquemas** agrupan técnicas, permitiendo definir nombres y grupos que identifiquen esas técnicas como *alto nivel de detalle*, *medio rendimiento*, etc.

15.4.1. Composición

Un material en Ogre está formado por una o varias *técnicas*, que contienen a su vez una o varias **Pasadas** (ver Figura 15.10). En cada momento sólo puede existir una *técnica* activa, de modo que el resto de técnicas no se emplearán en esa etapa de render.

Una vez que Ogre ha decidido qué *técnica* empleará, generará tantas pasadas (en el mismo orden de definición) como indique el material. Cada *pasada* define una operación de despliegue en la GPU, por lo que si una técnica tiene cuatro pasadas asociadas a un material tendrá que desplegar el objeto tres veces en cada *frame*.

Las **Unidades de Textura** (*Texture Unit*) contienen referencias a una única textura. Esta textura puede ser generada en código (*procedural*), puede ser obtenida mediante un archivo o mediante un flujo de vídeo. Cada *pasada* puede utilizar tantas *unidades de textura* como sean necesarias. Ogre optimiza el envío de texturas a la GPU, de modo que únicamente se descargará de la memoria de la tarjeta cuando no se vaya a utilizar más.

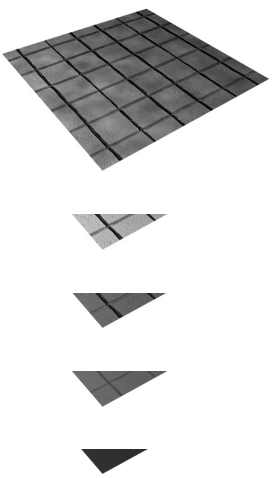


Figura 15.9: Un material en Ogre se describe mediante sucesivas pasadas que van configurando la apariencia final.

Atributo	Descripción
lod_values	Lista de distancias para aplicar diferentes niveles de detalle. Está relacionado con el campo *lod_strategy* (ver API de Ogre).
receive_shadows	Admite valores *on* (por defecto) y *off*. Indica si el objeto sólido puede recibir sombras. Los objetos transparentes *nunca* pueden recibir sombras (aunque sí arrojarlas, ver el siguiente campo).
transparency_casts_shadows	Indica si el material transparente puede arrojar sombras. Admite valores *on* y *off* (por defecto).
set_texture_alias	Permite crear alias de un nombre de textura. Primero se indica el nombre del alias y después del nombre de la textura original.

Cuadro 15.1: Atributos generales de *Materiales*.

Atributo	Formato	Descripción
ambient	r g b [a]	Valor de sombreado ambiente (por defecto 1.0 1.0 1.0 1.0).
diffuse	r g b [a]	Valor de sombreado difuso (por defecto 1.0 1.0 1.0 1.0).
specular	r g b [a] h	Valor de sombreado especular (por defecto 0.0 0.0 0.0 0.0). El valor de dureza (*shininess*) puede ser cualquier valor >0.
emissive	r g b [a]	Valor de emisión (por defecto 0.0 0.0 0.0 0.0).
scene_blend	(Ver valores)	Tipo de mezclado de esta pasada con el resto de la escena. Por defecto no se realiza mezclado. Si se especifica, puede tomar valores entre *add*, *modulate*, *colour_blend* y *alpha_blend*.
depth_check	on | off	Por defecto *on*. Indica si se utilizará el *depth-buffer* para comprobar la profundidad.
lighting	on | off	Por defecto *on*. Indica si se empleará iluminación dinámica en la pasada.
shading	(Ver valores)	Indica el tipo de método de interpolación de iluminación a nivel de vértice. Se especifican los valores de interpolación de sombreado *flat* o *gouraud* o *phong*. Por defecto se emplea el método de *gouraud*.
polygon_mode	(Ver valores)	Indica cómo se representarán los polígonos. Admite tres valores: *solid* (por defecto), *wireframe* o *points*.

Cuadro 15.2: Atributos generales de las *Pasadas*. Ver API de Ogre para una descripción completa de todos los atributos soportados.

15.4.2. Ejemplo de Materiales

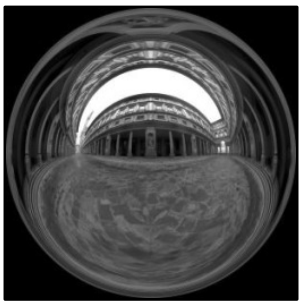


Figura 15.12: Mapa de entorno esférico empleado para simular la reflexión de un supuesto mundo.

A continuación veremos un ejemplo sencillo de definición de materiales en Ogre. Como hemos visto, el material definido en el listado de la Figura 15.11 contiene una técnica y una única pasada que define un método de sombreado difuso (especificando el color base en RGB). El nombre asignado al material especificado a la derecha de la etiqueta *Material* (en este caso "*Material1*") debe ser único a lo largo de la aplicación. El nombre de los *elementos* que definen el material (*técnicas*, *pasadas* y *unidades de textura*) es opcional. Si no se especifica ninguno, Ogre comenzará a nombrarlas comenzando en 0 según el orden de especificación del script. El nombre de estos *elementos* puede repetirse en diferentes materiales.

Veamos a continuación un ejemplo más complejo de definición de material. En la pasada definida se utiliza una *texture_unit* que referencia a un archivo de mapa de entorno esférico.

```
material Material1
{
  technique
  {
    pass
    {
      diffuse 0.5 0.5 0.5
    }
  }
}
```

Figura 15.11: Definición y resultado de un sencillo material con sombreado difuso.

```
material Material2
{
  technique
  {
    pass
    {
      diffuse 0.99 0.9 0.5
      ambient 0.2 0.2 0.2
      specular 1.0 1.0 0.9 30
      texture_unit
      {
        texture envmap.png
        env_map spherical
        colour_op modulate
      }
    }
  }
}
```

Figura 15.13: Un material más complejo que incluye la definición de una *texture_unit* con diversas componentes de sombreado.

Los mapas de entorno se utilizan para simular la reflexión del mundo (representado en el mapa) dependiendo de la relación entre las normales de las caras poligionales y la posición del observador. En este caso, se indica a Ogre que el tipo de mapa de entorno es esférico (tipo *ojo de pez*, ver Figura 15.12). El operador de *colour_op* empleado indica cómo se combinará el color de la textura con el color base del objeto. Existen diversas alternativas; mediante *modulate* se indica a Ogre que *multiplique* el color base (en este caso, un color amarillo indicado en la componente difusa del color) y el color del mapa.

A continuación se describen algunas de las propiedades generales de los materiales. En la tabla 15.1 se resumen los atributos generales más relevantes empleados en la definición de materiales, mientras que las tablas 15.2 y 15.3 resumen los atributos globales más utilizados en la definición de las pasadas y unidades de textura resepectivamente. Cada pasada tiene asociada cero o más unidades de textura. En los siguientes ejemplos veremos cómo combinar varias pasadas y unidades de textura para definir materiales complejos en Ogre.

Atributo	Descripción
texture	Especifica el nombre de la textura (estática) para esta *texture_unit*. Permite especificar el tipo, y el uso o no de canal alpha separado.
anim_texture	Permite utilizar un conjunto de imágenes como textura animada. Se especifica el número de frames y el tiempo (en segundos).
filtering	Filtro empleado para ampliar o reducir la textura. Admite valores entre *none*, *bilinear* (por defecto), *trilinear* o *anisotropic*.
colour_op	Permite determinar cómo se mezcla la *unidad de textura*. Admite valores entre *replace*, *add*, *modulate* (por defecto) o *alpha_blend*.
colour_op_ex	Versión extendida del atributo anterior, que permite especificar con mucho mayor detalle el tipo de mezclado.
env_map	Uso de mapa de entorno. Si se especifica (por defecto está en *off*, puede tomar valores entre *spherical*, *planar*, *cubic_reflection* y *cubic_normal*.
rotate	Permite ajustar la rotación de la textura. Requiere como parámetro el ángulo de rotación (en contra de las agujas del reloj).
scale	Ajusta la escala de la textura, especificando dos factores (en X e Y).

Cuadro 15.3: Atributos generales de las *Unidades de Textura*. Ver API de Ogre para una descripción completa de todos los atributos soportados

15.5. Mapeado UV en Blender

Mapas UV

El mapeado UV es el estándar en desarrollo de videojuegos, por lo que prestaremos especial atención en este documento.

La forma más flexible para la proyección de texturas en aplicaciones interactivas es el mapeado paramétrico UV. Como hemos visto, a cada vértice del modelo (con coordenadas *X,Y,Z*) se le asocian dos coordenadas paramétricas 2D (*U,V*).

Los modelos de mapeado ortogonales no se ajustan bien en objetos complejos. Por ejemplo, la textura asociada a una cabeza de un personaje no se podría mapear adecuadamente empleando modelos de proyección ortogonal. Para tener el realismo necesario en modelos pintados manualmente (o proyectando texturas basadas en fotografías) es necesario tener control sobre la posición final de cada píxel sobre la textura. El mapa UV describe qué parte de la textura se asociará a cada polígono del modelo, de forma que, mediante una operación de *despliegue* (*unwrap*) del modelo obtenemos la equivalencia de la superficie en el plano 2D.

En esta sección estudiaremos con más detalle las opciones de Blender para trabajar con este tipo de coordenadas.

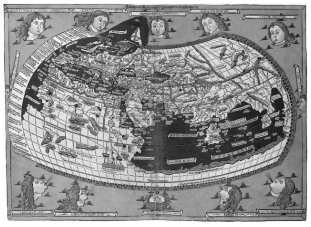


Figura 15.14: El problema del *despliegue* se ha afrontado desde los orígenes de la cartografía. En la imagen, un mapa de 1482 muestra una proyección del mundo conocido hasta el momento.

Como vimos en el capítulo 11, para comenzar es necesario aplicar una operación de *despliegue* del modelo para obtener una o varias regiones en la ventana de UV/Image Editor . Esta operación de despliegue se realiza siempre en modo edición. Con el objeto en modo edición, en los botones de edición , y tras aplicar el despliegue (*Unwrap*) (tecla U), aparece un nuevo subpanel en el panel de herramientas (*Tool Shelf*), accesible mediante la tecla T (ver Figura 15.15) que controla el modo en el que se realiza el *despliegue* del modelo. Este panel aparece mientras el objeto está en modo de edición, y es dependiente del modo de despliegue elegido (por ejemplo, el de la Figura 15.15 contiene las opciones de *Unwrap*, pero otros modos de despliegue tendrán otras opciones asociadas). A continuación describimos las principales opciones que pueden encontrarse en este tipo de subpaneles:

Isla UV

Se define una isla en un mapa UV como cada región que contiene vértices no conectados con el resto.

- **Angle Based | Conformal**. Define el método de cálculo de la proyección. El basado en ángulo crea una nueva isla cuando el ángulo entre caras vecinas sea relevante. En el modo *conformal* se realiza dependiendo del método de proyección seleccionado. El método basado en ángulo ofrece, en general, mejores resultados.

- **Fill Holes**. Intenta ordenar todas las islas para evitar que queden huecos en la textura sin ninguna cara UV.
- **Correct Aspect**. Permite el escalado de los vértices desplegados en UV para ajustarse a la relación de aspecto de la textura a utilizar.
- **Margin**. Define la separación mínima entre islas.
- **Use Subsurf Data**. Utiliza el nivel de subdivisión especificado en **Subsurf Target** para el cálculo de las coordenadas UV en lugar de la malla original (Red de Control).
- **Transform Correction**. Corrige la distorsión del mapa UV mientras se está editando.
- **Cube Size**. Cuando se emplea una proyección cúbica, este parámetro indica el porcentaje de la imagen que se emlpeará por el mapa UV (por defecto el 100 %).
- **Radius**. Cuando se emplea el método de proyección *Cylinder projection*, este parámetro define el radio del cilindro de proyección. A menores valores, las coordenadas del objeto aparecerán estiradas en el eje Y.
- **Align: Polar ZX | Polar ZY**. Determina el plano frontal de proyección en los métodos cilíndrico y esférico.

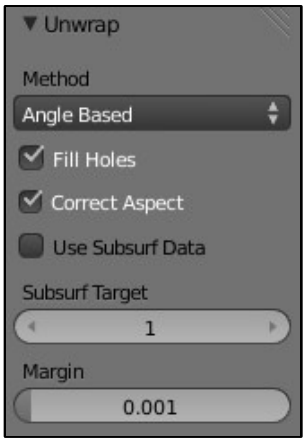


Figura 15.15: Opciones del panel *Unwrap*.

El modo general de trabajo asociado al despliegue de una malla está compuesto por la siguiente serie de pasos secuenciales:

1. Seleccionar el objeto que queremos desplegar.
2. Suele ser interesante ver los cambios realizados sobre la textura UV cambiando el modo de dibujado del objeto en la ventana 3D a Texture .
3. Cambiar al modo edición (mediante (TAB)) y seleccionar las caras que queremos desplegar. En algunos casos serán todas (tecla (A)), o puede ser conveniente elegir individualmente (en este caso es conveniente elegir el modo de selección de caras en la cabecera de la ventana 3D).
4. Establecer los valores globales de despliegue en la pestaña asociada al método de despliegue elegido (ver Figura 15.15).
5. Verificar que existe al menos una capa de textura UV en el panel Mesh del grupo de botones *Object Data* (ver Figura 15.16). Cada cara de la malla puede tener varias texturas UV, pero cada textura únicamente puede tener una única imagen asignada.
6. Pulsando la tecla (U) accedemos al menú de despliegue (*UV Mapping*) que nos permite elegir el método de despiegue que aplicaremos as la malla. Entre estos métodos podemos destacar los siguientes (ver Figura 15.19):
   - **Unwrap**. Esta opción despliega las caras del objeto tratando de obtener una configuración que rellene de la mejor forma posible la textura, empleando información topológica de la malla (cómo están conectadas las caras de la malla). Si es posible, cada cara desplegada tendrá su propia región de la imagen sin solapar con otras caras.

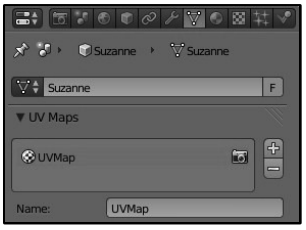


Figura 15.16: Opciones del panel *Mesh*.

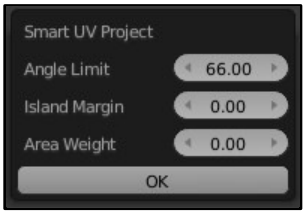


Figura 15.17: Configuración de las opciones de despliegue en *Smart UV Project*.

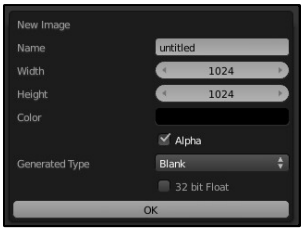


Figura 15.18: Ventana para la creación de una nueva textura para el mapeado UV.

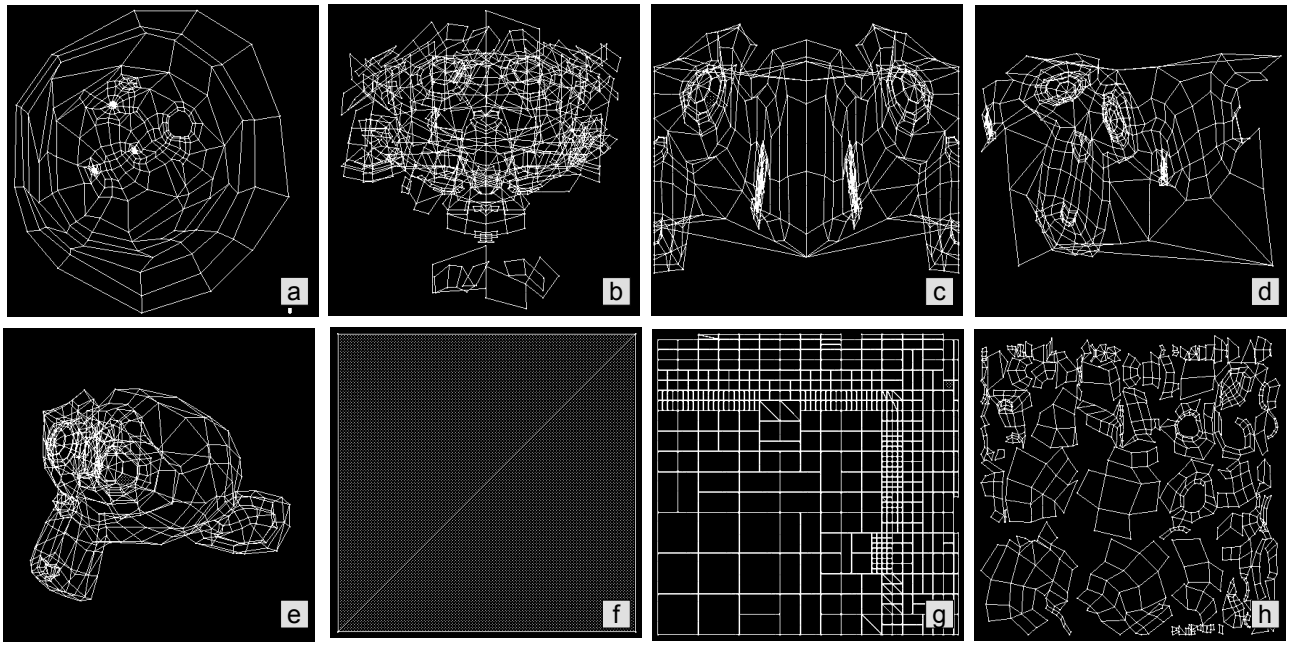


Figura 15.19: Resultado obtenido tras aplicar diferentes métodos de despliegue (*unwrap*) al modelo de Suzanne con todas las caras seleccionadas: **a)** Unwrap, **b)** Cube, **c)** Cylinder, **d)** Sphere, **e)** Project from view, **f)** Reset, **g)** Lightmap y **h)** Smart UV.

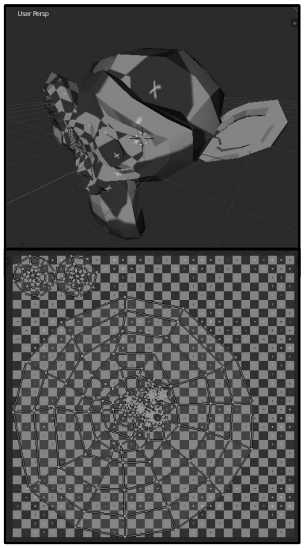


Figura 15.20: Resultado de aplicar la textura de prueba a la malla de *Suzanne* con el modo de despliegue básico.

- **Smart UV Projects**. Este método estudia la forma del objeto, estudiando la relación entre las caras seleccionadas y crea un mapa UV basada en las opciones de configuración que se muestran en la Figura 15.17. A menor límite de ángulo, mayor número de islas se crearán. El margen existente entre islas se puede elegir en el parámetro *Island Margin*. El parámetro de *Area Weight* permite dar mayor peso a las caras de mayor superficie. Este método de proyección automático se ajusta perfectamente en multitud de situaciones, siendo el que probablemente ofrezca una mejor configuración de partida para ajustar posteriormente los vértices de forma manual.
- **Lightmap Pack**. Despliega las caras de forma regular para ser utilizados en el cálculo de mapas de luz. Este tipo de despiegue será utilizado en el Capítulo 16 para el precálculo de iluminación.
- **Follow Active Quads**. Este método toma como entrada las caras seleccionadas y las despliega siguiendo bucles de cara continuos. Este método no tiene en cuenta el tamaño de la imagen.
- **Cube | Cylinder | Sphere Projection**. Estos métodos tratan de despegar el objeto empleando estos modelos de proyección ortogonales. Como hemos visto anteriormente, en estos métodos resulta especialmente relevante la definición de las opciones del panel de la Figura 15.15.
- **Project from View | (Bounds)**. Emplea el punto de vista 3D para desplegar el objeto. Si se elige la opción *Bounds*, el despliegue se realizará ajustando a los límites de la imagen.
- **Reset**. Mediante esta opción todas las caras se despliegan ocupando el 100 % de la imagen, con vértices en las esquinas de la misma.

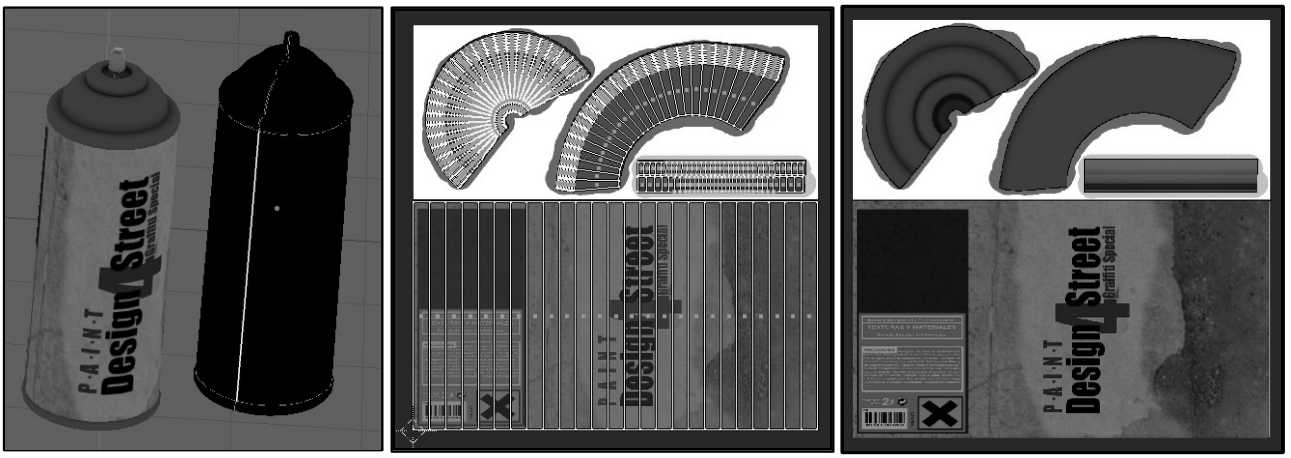


Figura 15.23: Utilización de *costuras* y despliegue del modelo utilizando diferentes técnicas de despliegue. En la imagen de la izquerda se marcan perfectamente los *seams* definidos para recortar el modelo adecuadamente.

Una vez que la malla ha sido desplegada, podemos ayudarnos de la herramienta de generación de rejillas de prueba que trae integrado Blender para hacernos una idea del resultado del despliegue. En la cabecera de la ventana del editor UV , accediendo al menú *Image/ New Image* la ventana de la Figura 15.18, donde podemos elegir la resolución en píxeles de la nueva imagen (por defecto 1024x1024 píxeles), el color de fondo y el nivel de Alpha. Si activamos el botón *UV Test Grid*, se creará la imagen con la rejilla de fondo que se muestra en la Figura 15.20. Esta imagen permite hacerse una idea del resultado del despliegue tras aplicar la textura al objeto 3D.

Las coordenadas UV se almacenan en el modelo de Blender y son pasadas a Ogre empleando el script de exportación. Sin embargo, es común utilizar un editor de imágenes externo para asignar colores a la textura. Para guardar el estado del despliegue (y utilizarlo como plantilla para el programa de edición, como GIMP (GNU Image Manipulation Program)), es posible emplear el script de la cabecera de la ventana del *UV/ Image Editor* en *UVs / Export UV Layout* para generar una imagen PNG, EPS o SVG.

En la cabecera de la ventana 3D puede elegirse el modo *Texture Paint* (ver Figura 15.21). Cuando este modo está activo, nos permite dibujar directamente sobre el modelo 3D, accediendo a los controles que se encuentran en el subpanel de la vista 3D accesible medidante la tecla T ((ver Figura 15.22)).

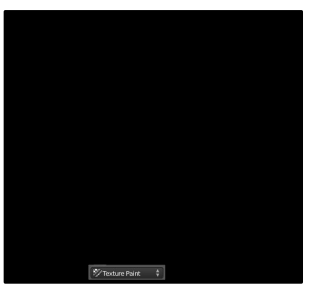


Figura 15.21: Editando la textura aplicada a Suzanne en el espacio 3D mediante el modo *Texture Paint*.

15.5.1. Costuras

En el despliegue de mallas complejas, es necesario ayudar a los métodos de despliegue definiendo costuras (*seams*). Estas costuras servirán para definir las zonas de recorte, guiando así al método de desplegado.

Para definir una costura basta con elegir las aristas que la definen en modo edición, tal y como se muestra en la Figura 15.23 . La selección de bucles de aristas (como en el caso del bote de spray del modelo) puede realizarse cómodamente seleccionando una de las aristas del bucle y empleando la combinación de teclas Control E *Edge Loop*. Cuando tenemos la zona de recorte seleccionada, definiremos la costura mediante la combinación Control E *Mark Seam*. En este menú aparece igualmente la opción para eliminar una costura *Clear Seam*.

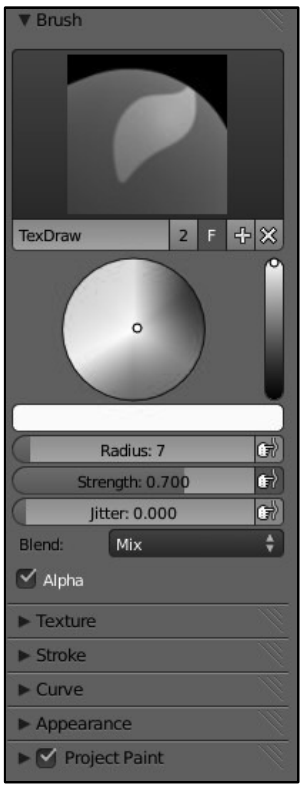


Figura 15.22: Subpanel de opciones de *Texture Paint* en el que puede elegirse el color de pintado, tamaño del pincel, opacidad, etc...

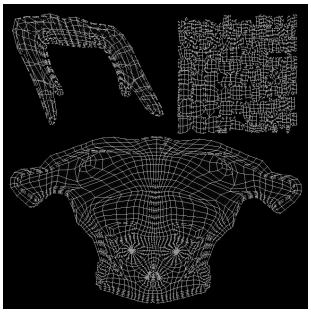


Figura 15.25: Despliegue del modelo de los ejemplos. Se ha empleado una operación de despliegue cinlíndrico para la mandíbula inferior, *Smart Projections* para los dientes y el *Unwrap* básico para el resto del objeto.

Una vez definidas las costuras, podemos emplear la selección de zonas enlazadas (mediante la tecla L) para elegir regiones conectadas. Podemos aplicar a cada grupo de caras un modo de despliegue distinto. Por ejemplo, en la Figura 15.23 se ha utilizado una proyección cilíndrica para el cuerpo central del Spray y para el difusor, mientras que la base y la zona superior se han desplegado mediante el modo general Unwrap.

Cada zona de despliegue en la ventana de *UV Image Editor* puede igualmente desplazarse empleando los operadores habituales de traslación G, rotación R y escalado S. En esta ventana puede igualmente emplearse el atajo de teclado para seleccionar los vértices conectados (link) L y seleccionar rápidamente regiones desplegadas (incrementalmente si pulsamos L mientras mantenemos pulsada la tecla Shift).

15.6. Ejemplos de Materiales en Ogre

En esta sección estudiaremos algunos ejemplos de definición de materiales empleando diversas unidades de tetura en Ogre. Nos centraremos en el uso de algunos operadores para la composición de diferentes unidades de textura. Las Figuras 15.24 y 15.27 muestran el resultado de definición de estos materiales y texturas. Veamos algunas de las propiedades empleadas en su definición.

En la definición del **Material3** se ha empleado el despliegue que se muestra en la Figura 15.25. Este material simplemente utiliza las coordenadas UV del modelo para proyectar la misma textura que en el suelo de la escena.

En la definición del **Material4** se utiliza una textura de tipo *cebra* donde las bandas negras son totalmente transparentes. En la pasada de este material se utiliza el atributo *scene_blend* que permite especificar el tipo de mezclado de esta pasada con el resto de contenido de la escena. Como veremos en el ejemplo del *Material6*, la pricipal diferencia entre las operaciones de mezclado a nivel de pasada o a nivel de textura es que las primeras definen la mezcla con el contenido *global* de la escena, mientras que las segundas están limitadas entre capas de textura. Mediante el modificador *alpha_blend* se indica que utilizaremos el valor *alpha* de la textura. A continuación estudiaremos las opciones permitidas por el atributo *scene_blend* en sus dos versiones.

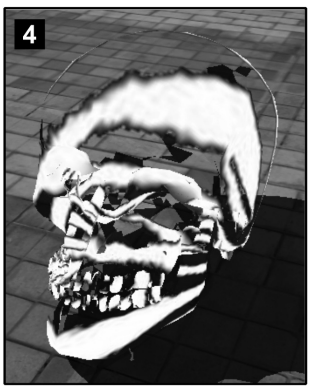
El formato de **scene_blend** permite elegir cuatro parámetros en su forma básica:

- **add**. El color del resultado de la pasada se *suma* al resultado de la escena.
- **modulate**. El resultado de la pasada se *multiplica* con el resultado de color de la escena.
- **colour_blend**. El color resultado de la pasada se mezcla empleando la componente de brillo de los colores.
- **alpha_blend**. Este modo utiliza la información de transparencia del resultado.

El atributo **scene_blend** permite otro formato mucho más general y flexible, que requiere dos parámetros: *scene_blend src dest*. En esta segunda versión, el color final se obtiene como: *c = (texture * src) + (scene_pixel * dest)*. En esta segunda versión, *src* y *dest* son factores de mezclado, entre 10 posibles valores:

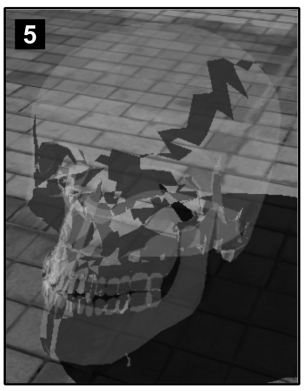
- *one*. Valor constante 1.0.
- *zero*. Valor constante 0.0.

```
material Material3
{
 technique
 {
  pass
  {
   diffuse 0.5 0.5 0.5
   texture_unit
   {
    texture grid.jpg
   }
  }
 }
}
```




```
material Material4
{
 technique
 {
  pass
  {
   scene_blend
     alpha_blend
   depth_write off
   lighting off

   texture_unit
   {
     texture
       zebratrp.png
   }
  }
 }
}
```




```
material Material5
{
 technique
 {
  pass
  {
   lighting off
   scene_blend
      alpha_blend
   depth_write off

   texture_unit
   {
    colour_op_ex
      source1
      src_manual
      src_current
      0.0 1.0 0.0
    alpha_op_ex
      source1
      src_manual
      src_current 0.25
}}}}}
```

```
material Material6
{
 technique
 {
 pass
 {
  texture_unit
  {
   texture grass.jpg
  }
  texture_unit
  {
   texture zebratrp.png
   colour_op alpha_blend
  }
  texture_unit
  {
    texture rock.jpg
    colour_op_ex
    blend_current_alpha
    src_texture
       src_current
}}}}
```

Figura 15.24: Ejemplos de definición de algunos materiales empleando diversos modos de composición en Ogre.

- *dest_colour*. Color del píxel en la escena.
- *src_colour*. Color del téxel (de la pasada).
- *one_minus_dest_colour*. 1 - (dest_colour).
- *one_minus_src_colour*. 1 - (src_colour).
- *dest_alpha*. Valor alfa del píxel en la escena.
- *src_alpha*. Valor alfa del téxel (de la pasada).
- *one_minus_dest_alpha*. 1 - (dest_alpha).
- *one_minus_src_alpha*. 1 - (src_alpha).

De este modo, el *scene_blend* por defecto que se aplica es el totalmente opaco, utilizando totalmente el valor del téxel de la pasada y sin tener en cuenta el valor del píxel existente en la escena (*scene_blend one zero*). Análogamente, el equivalente al modo básico de *alpha_blend* estudiado anteriormente sería *scene_blend src_alpha one_minus_src_alpha*, el modo básico *add* se escribiría como *scene_blend one one*, etc...

Figura 15.26: En los ejemplos se han utilizado dos versiones de la textura de cebra; una con transparencia total definida en las bandas negras (en formato PNG), y otra en JPG sin transparencia.

Consulta el manual!

En el manual de Ogre pueden consultarse todos los parámetros que admiten todos los operadores disponibles a nivel de textura y pasadas.

Para finalizar la descripción de los atributos utilizados en el *Material4*, mediante *depth_write off* se fuerza a que la pasada no utilice el *ZBuffer*. Es habitual desactivar el *ZBuffer* cuando se despliegan objetos transparentes sobre la escena, para que se superpongan adecuadamente. El atributo *lighting off* indica que no se tenga en cuenta la iluminación dinámica en esta pasada. Cuando se desactiva la iluminación dinámica, las propiedades de sombreado difusas, ambientales, especulares y de emisión no se tienen en cuenta, por lo que cualquier definición en el material sería redundante.

El **Material5** define un color transparente definiendo una unidad de textura manualmente. Este material hace uso de las versiones que más precisión ofrecen a la hora de definir cómo se combina el color (y la opacidad) de una capa de textura con las anteriores. Así, la definición del color manualmente de la fuente como verde, y el nivel de transparencia como 0.25. En términos generales, el atributo *colour_op_ex* requiere como primer parámetro el identificador de una operación *operation*, y luego dos fuentes *src1 src2*. Las fuentes pueden ser una de las siguientes cinco opciones:

- *src_current*. El color obtenido en capas anteriores.
- *src_texture*. El color de la capa actual.
- *src_diffuse*. El color difuso de los vértices.
- *src_specular*. El color especular de los vértices.
- *src_manual*. Definición manual del color.

Como operación admite 15 valores diferentes. Si se indica *source1* o *source2* se utilizará su valor directamente sin modificación. La operación *blend_current_alpha* (como veremos en el siguiente ejemplo) utiliza la información de *alpha* de las capas anteriores. La operación *modulate* multiplica los valores de *src1* y *src2*.

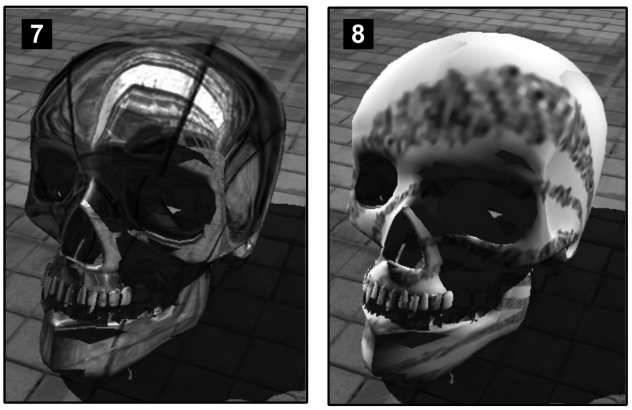
El **Material6** define tres capas de textura. La primera carga una textura de césped. La segunda capa carga la textura de cebra con transparencia, e indica que la combinará con la anterior empleando la información de alpha de esta segunda textura. Finalmente la tercera capa utiliza el operador extendido de *colour_op_ex*, donde define que combinará el color de la capa actual (indicado como primer parámetro en *src_texture* con el obtenido en las capas anteriores *src_current*. El parámetro *blend_current_alpha* multiplica el alpha de *src2* por *(1-alpha(src1))*.

Los ejemplos de la Figura 15.27 utilizan algunas de las opciones de animación existentes en las unidades de textura. El **Material7** por ejemplo define dos capas de textura. La primera aplica la textura del suelo al objeto, utilizando el atributo *rotate_anim*. Este atributo requiere un parámetro que indica el número de revoluciones por segundo (empleando velocidad constante) de la textura.

La composición de la segunda capa de textura (que emplea un mapeado de entorno esférico) se realiza utilizando el atributo *colour_op_ex* estudiado anteriormente. La versión de la operación *modulate_x2* multiplica el resultado de *modulate* (multiplicación) por dos, para obtener un resultado más brillante.

En el ejemplo del **Material8** se definen dos capas de textura, ambas con animación. La segunda capa utiliza una operación de suma sobre la textura de *cebra* sin opacidad, de modo que las bandas negras se *ignoran*. A esta textura se aplica una rotación de 0.15 revoluciones por segundo. La primera capa utiliza el atributo *scroll_anim* que permite definir un desplazamiento constante de la textura en X (primer parámetro) y en Y (segundo parámetro). En este ejemplo la textura únicamente se desplaza en el eje X. De igual modo, la primera capa emplea *wave_xform* para definir una animación basado en una función de onda. En este caso se utiliza una función senoidal indicando los parámetros requeridos de base, frecuencia, fase y amplitud (ver el manual de Ogre para más detalles sobre el uso de la función).

```
material Material7
{
 technique
 {
  pass
  {
    ambient 1 1 1
    diffuse 1 1 1
    Specular 1 1 1 9800
    emissive 0 0 0 1
    texture_unit
    {
      rotate_anim 0.01
      texture grid.jpg
    }
    texture_unit
    {
      texture envmap.png
      colour_op_ex
        modulate_x2
        src_texture
        src_current
      env_map spherical
}}}}
```




```
material Material8
{
 technique
 {
  pass
  {
   ambient 0.5 0.5 0.5
   diffuse 1.0 1.0 1.0

   texture_unit
   {
     texture grass.jpg
     scroll_anim 0.1 0.0
     wave_xform
       scale sine
       0.0 0.2 0.0 0.2
   }

   texture_unit
   {
     texture zebra.jpg
     rotate_anim 0.15
     colour_op add
]}}}
```

Figura 15.27: Ejemplos de uso de animación de texturas (rotación y onda senoidal).

15.7. Render a Textura

En esta sección estudiaremos un ejemplo en el que se definirá una textura que contendrá el resultado de renderizar la escena empleando otra cámara auxiliar. Para ello, estudiaremos la creación manual de texturas, y el uso de un *Listener* particular que será ejecutado cada vez que se produzca una actualización en la textura.

El objeto sobre el que se proyectará la textura está definido en la Figura 15.28.a). En este modelo se han definido tres materiales, cada uno con sus propias coordenadas de despiegue. Las coordenadas más importantes son las relativas a la pantalla de la televisión, sobre la que desplegaremos la textura. Estas coordenadas se han definido de modo que ocupan todo el área de mapeado (como se muestra en la Figura 15.28.b). El material asociado a la pantalla se ha nombrado como *"pantallaTV"*, y será editado en código a nivel de *SubEntity*. Cuando un modelo cuenta con diversos materiales (como es el caso), Ogre crea un objeto de la clase *SubEntity* para cada material, de modo que, como veremos a continuación, tendremos que acceder a todos los *SubEntity* del objeto para elegir el que tiene el material de la pantalla y cambiarlo por el que calcularemos en tiempo de ejecución.

Hasta ahora hemos desplegado el resultado de la escena sobre la ventana principal de la aplicación. Sin embargo, en multitud de aplicaciones es habitual renderizar la escena total o parcialmente sobre una textura. Ogre facilita enormemente la construcción de este tipo de texturas. Una vez que tenemos la textura, podemos aplicar cualquier operador de los estudiados en la sección anterior para mezclarlas con otras capas de textura.

El siguiente listado muestra el código relevante para el ejemplo de Render a Textura.

El primer paso para aplicar el Render a Textura es obtener un objeto de tipo *TexturePtr* (líneas 1-3), que permite crear una textura manualmente, indicando el nombre de la textura ("*RttT*"), y las propiedades de tamaño (*512x512* píxeles), así como el formato de color (32Bits en RGB, sin canal alfa *PF_R8G8B8*). El último parámetro de *TU_RENDERTARGET* indica a Ogre el tipo de uso que haremos de la textura.

Render a Textura

Existen multitud de aplicaciones en videojuegos para utilizar Render a Textura. Por ejemplo, los espejos de cualquier simulador de conducción o cámaras de vigilancia dentro del juego, así como multitud de efectos gráficos (como espejos, motion blur...) y de postproducción que se realizan empleando esta técnica.

Figura 15.28: Construcción del ejemplo de RTT. **a)** Modelo empleado para desplegar la textura de RTT. El modelo se ha definido con tres materiales; uno para la carcasa, otro para las letras del logotipo (ambos desplegados en **c)**, y uno para la pantalla llamado "*pantallaTV*", desplegado en **b)**. En **d)** se muestra el resultado de la ejecución del programa de ejemplo.

Listado 15.1: Fragmento de CreateScene (MyApp.cpp).

```
1  TexturePtr rtt = TextureManager::getSingleton().createManual(
2   "RttT", ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
3   TEX_TYPE_2D, 512, 512, 0, PF_R8G8B8, TU_RENDERTARGET);
4
5  RenderTexture *rtex = rtt->getBuffer()->getRenderTarget();
6
7  Camera *cam = _sceneManager->createCamera("SecondCamera");
8  cam->setPosition(Vector3(17,16,-4));
9  cam->lookAt(Vector3(-3,2.7,0));
10 cam->setNearClipDistance(5);
11 cam->setFOVy(Degree(38));
12
13 rtex->addViewport(cam);
14 rtex->getViewport(0)->setClearEveryFrame(true);
15 rtex->getViewport(0)->setBackgroundColour(ColourValue::Black);
16 rtex->getViewport(0)->setOverlaysEnabled(false);
17 rtex->setAutoUpdated(true);
18
19 MaterialPtr mPtr = MaterialManager::getSingleton().create(
20  "RttMat",Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
21 Technique* matTechnique = mPtr->createTechnique();
22 matTechnique->createPass();
23 mPtr->getTechnique(0)->getPass(0)->setLightingEnabled(true);
24 mPtr->getTechnique(0)->getPass(0)->setDiffuse(.9,.9,.9,1);
25 mPtr->getTechnique(0)->getPass(0)->setSelfIllumination(.4,.4,.4);
26
27 mPtr->getTechnique(0)->getPass(0)->createTextureUnitState("RttT");
28
29 for (unsigned int i=0; i<entTV->getNumSubEntities(); i++) {
30   SubEntity *aux = entTV->getSubEntity(i);
31   if (aux->getMaterialName() == "pantallaTV")
32     aux->setMaterialName("RttMat");
```

```
33 }
```

En la línea [5] se obtiene un puntero a un *RenderTexture* que es una especialización de la clase *RenderTarget* específica para renderizar sobre una textura.

Puede haber varios *RenderTargets* que generen resultados sobre la misma textura. A este objeto de tipo *RenderTexture* le asociamos una cámara en la línea [13] (que ha sido creada en las líneas [7-11]), y configuramos las propiedades específidas del viewport asociado (como que se limpie en cada frame en la línea [14], y que no represente los overlays en la línea [16]).

En la línea [17] indicamos que la textura se actualice automáticamente (gestionada por el bucle principal de Ogre). En otro caso, será necesario ejecutar manualmente el método *update* del *RenderTarget*.

Las líneas [19-25] declaran manualmente el material sobre el que emplearemos la *Texture Unit* con la textura anteriormente definida. En [19-20] creamos un material llamado "*RttMat*", con una única técnica (línea [21]) y una pasada (línea [22]). Esa pasada define sus propiedades en las líneas [23-25], y añade como *TextureUnit* a la textura "*RttT*".

El último bucle recorre todas las *SubEntities* que forman a la entidad de la televisión. En el caso de que el material asociado a la subentidad sea el llamado "*pantallaTV*" (línea [31]), le asignamos el material que hemos creado anteriormente (línea [32]).

15.7.1. Texture Listener

En el ejemplo anterior, la textura proyectada sobre la televisión tiene un efecto recursivo debido a que aparece en el render de la cámara auxiliar. En muchos casos, interesa ocultar uno o varios objetos de la escena para realizar el render a textura (por ejemplo si queremos simular el punto de vista desde el interior de un objeto).

Al igual que ocurre a nivel de *Frame*, es posible añadir uno o varios objetos *Listener* para controlar la actualización de las texturas. De este modo, cada vez que se renderiza un *RenderTarget* (en nuestro caso concreto una textura), Ogre invocará previamente el método asociado al *preRenderTargetUpdate*. Cuando la textura se haya actualizado, se ejecutará el método llamado *postRenderTargetUpdate*.

El siguiente listado muestra el fichero de cabecera de la declaración de una clase propia llamada *MyTextureListener* que implementa el interfaz definido en *RenderTargetListener*. Esta clase recibe en el constructor un parámetro con el puntero a un objeto de tipo *Entity*. La clase se encargará de ocultar esa entidad antes de renderizar la escena sobre la textura y de volver a mostrarlo tras el despliegue.

Listado 15.2: MyTextureListener.h

```
1  #include <Ogre.h>
2  using namespace std;
3  using namespace Ogre;
4
5  class MyTextureListener : public RenderTargetListener {
6  private:
7   Entity* _ent;
8  public:
9   MyTextureListener(Entity *ent);
10  ~MyTextureListener();
11  virtual void preRenderTargetUpdate(const RenderTargetEvent& evt);
12  virtual void postRenderTargetUpdate(const RenderTargetEvent& evt);
13 };
```

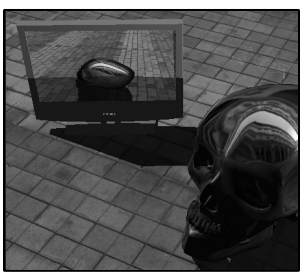


Figura 15.29: Tras aplicar el textu-re listener que oculta la entidad de la televisión, se consigue eliminar el efecto de despliegue recursivo.

A continuación se muestra el listado que define la clase MyTextureListener. La funcionalidad ha sido comentada anteriormente.

Listado 15.3: MyTextureListener.cpp

```
1  #include "MyTextureListener.h"
2
3  MyTextureListener::MyTextureListener(Entity* ent){
4    _ent = ent;
5  }
6
7  MyTextureListener::~MyTextureListener(){ }
8
9  void MyTextureListener::preRenderTargetUpdate(const
       RenderTargetEvent& evt) {
10   cout << "preRenderTargetupdate" << endl;
11   _ent->setVisible(false);
12 }
13
14 void MyTextureListener::postRenderTargetUpdate(const
       RenderTargetEvent& evt) {
15   cout << "postRenderTargetupdate" << endl;
16   _ent->setVisible(true);
17 }
```

El uso de la clase es muy sencillo. Basta con añadir el listener al objeto de tipo *RenderTexture* (ver línea [2] del siguiente listado).

Listado 15.4: Utilización en MyApp.cpp

```
1 _textureListener = new MyTextureListener(entTV);
2 rtex->addListener(_textureListener);
```

C15

15.7.2. Espejo (Mirror)

La reflexión en *Espejo* es otro de los efectos clásicos que se obtienen mediante render a textura. El siguiente listado muestra el código necesario para realizar este ejemplo. Estudiaremos los aspectos que lo distinguen sobre el código de la sección anterior.

En las líneas [8-12] se define la cámara que utilizaremos para crear el efecto de *mirror*. Esta cámara *debe* tener la misma posición y orientación que la cámara desde donde percibimos la escena, pero debe ser independiente (ya que activaremos la reflexión sobre un plano, y modificaremos su plano de recorte cercano en las líneas *42-43*). Esta cámara para el efecto de espejo (llamada *MirrorCamera* en este ejemplo) debe estar *siempre* correctamente alineada con la cámara principal. En este ejemplo, la cámara principal es estática, por lo que tampoco modificaremos la posición y orientación de la *MirrorCamera*. Queda como ejercicio propuesto para el lector añadir movimiento a la cámara principal, actualizando análogamente en cada frame la posición de la cámara *Mirror*.

Figura 15.30: Resultado de aplicar el material con Render a Textura para simular espejo sobre el plano del suelo.

En las líneas [24-30] se definen las propiedades generales del material. La línea [31] nos permite generar las coordenadas de textura según la cámara que se le pasa como argumento, de modo que da la impresión de que la textura ha sido proyectada sobre la superficie. De esta forma, la textura que generamos será posteriormente renderizada utilizando el modelo de proyección definido por la cámara de *Mirror*.

Para concluir, en las líneas 42-43 se configuran los aspectos relativos a la cámara *Mirror* en relación al plano de reflexión. La llamada a *enableReflection* hace que se modifique el *Frustum* de la cámara de modo que renderice empleando la reflexión con respecto del plano que se le pasa como argumento.

Finalmente, la llamada a *enableCustomNearClipPlane* permite recortar la geometría situada *debajo* del plano pasado como argumento, de modo que únicamente la geometría que está situada *sobre* el plano será finalmente desplegada en el reflejo, evitando así errores de visualización.

Listado 15.5: Definición del Material tipo "Espejo"

```
1  TexturePtr rttM_texture = TextureManager::getSingleton()
2   .createManual("RttMTex", ResourceGroupManager::
3   DEFAULT_RESOURCE_GROUP_NAME, TEX_TYPE_2D, 512, 512, 0, PF_R8G8B8,
4   TU_RENDERTARGET);
5
6  RenderTexture *rMtex= rttM_texture->getBuffer()->getRenderTarget();
7
8  Camera *camM = _sceneManager->createCamera("MirrorCamera");
9  Camera *mainCam = _sceneManager->getCamera("MainCamera");
10 camM->setPosition(mainCam->getPosition());
11 camM->setOrientation(mainCam->getOrientation());
12 camM->setAspectRatio(mainCam->getAspectRatio());
13
14 rMtex->addViewport(camM);
15 rMtex->getViewport(0)->setClearEveryFrame(true);
16 rMtex->getViewport(0)->setBackgroundColour(ColourValue::Black);
17 rMtex->getViewport(0)->setOverlaysEnabled(false);
18 rMtex->setAutoUpdated(true);
19
20 MaterialPtr mMPtr=MaterialManager::getSingleton().create("RttMMat",
21   Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
22 Technique* matMTechnique = mMPtr->createTechnique();
23 matMTechnique->createPass();
24 TextureUnitState *t = mMPtr->getTechnique(0)->getPass(0)->
       createTextureUnitState("grid.jpg");
25 t = mMPtr->getTechnique(0)->
26     getPass(0)->createTextureUnitState("RttMTex");
27 t->setColourOperationEx(LBX_BLEND_MANUAL, LBS_TEXTURE,
28     LBS_CURRENT, ColourValue::White, ColourValue::White, 0.5);
29
30 t->setTextureAddressingMode(TextureUnitState::TAM_CLAMP);
31 t->setProjectiveTexturing(true, camM);
32
33 // Creacion del plano del suelo...
34 Plane plane1(Vector3::UNIT_Y, 0);
35 MeshManager::getSingleton().createPlane("plane1",
36  ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane1,
37  200,200,1,1,true,1,10,10,Vector3::UNIT_Z);
38
39 SceneNode* node3 = _sceneManager->createSceneNode("ground");
40 Entity* grEnt = _sceneManager->createEntity("planeEnt", "plane1");
41
42 camM->enableReflection(plane1);
43 camM->enableCustomNearClipPlane(plane1);
44
45 grEnt->setMaterialName("RttMMat");
```

Capítulo 16
Iluminación

Carlos González Morcillo

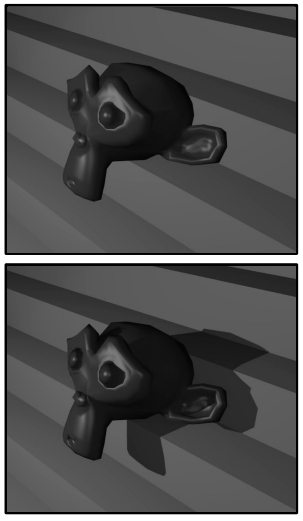


Figura 16.1: Gracias a la proyección de sombras es posible conocer la posición relativa entre objetos. En la imagen superior no es posible determinar si el modelo de *Suzanne* reposa sobre algún escalón o está flotando en el aire. La imagen inferior, gracias al uso de sombras elimina esa ambigüedad visual.

En el capítulo anterior hemos estudiado algunas de las características relativas a la definición de materiales básicos. Estas propiedades están directamente relacionadas con el modelo de iluminación y la simulación de las fuentes de luz que realicemos. Este capítulo introduce algunos conceptos generales sobre iluminación en videojuegos, así como técnicas ampliamente utilizadas para la simulación de la iluminación global.

16.1. Introducción

Una pequeña parte de los rayos de luz son visibles al ojo humano. Aquellos rayos que están definidos por una onda con longitud de onda λ entre 700 y 400nm. Variando las longitudes de onda obtenemos diversos colores.

En gráficos por computador es habitual emplear los denominados *colores-luz*, donde el Rojo, Verde y Azul son los colores primarios y el resto se obtienen de su combinación. En los *colores-luz* el color blanco se obtiene como la suma de los tres colores básicos.

El RGB es un modelo clásico de este tipo. En el mundo físico real se trabaja con *colores-pigmento*, donde el Cyan, el Magenta y el Amarillo forman los colores primerarios. La combinación de igual cantidad de los tres colores primarios obtiene el color negro[1]. Así, el CMYK (Cyan Magenta Yellow Key) empleado por las impresoras es un clásico modelo de este tipo. De un modo simplificado, podemos definir los pasos más relevantes para realizar la representación de una escena sintética:

[1]En realidad se obtiene un tono *parduzco*, por lo que habitualmente es necesario incorporar el negro como color primario.

1. La luz es emitida por las fuentes de luz de la escena (como el sol, una lámpara de luz situada encima de la mesa, o un panel luminoso en el techo de la habitación).
2. Los rayos de luz interactúan con los objetos de la escena. Dependiendo de las propiedades del material de estos objetos, parte de la luz será absorbida y otra parte reflejada y propagada en diversas direcciones. Todos los rayos que no son totalmente absorbidos continuarán rebotando en el entorno.
3. Finalmente algunos rayos de luz serán capturados por un sensor (como un ojo humano, el sensor CCD (Charge-Coupled Device) de una cámara digital o una película fotográfica).

La luz puede ser modelada empleando diversas aproximaciones, centrándose en las propiedades direccionales (rayos puramente geométricos), como ondas electromagnéticas o como partículas cuánticas (fotones). Dependiendo del método de representación, suele emplearse un modelo u otro. Independientemente del tratamiento que demos a la luz, ésta debe ser simulada como energía que *viaja* en el espacio. Las fuentes de luz serán *emisores* de esta energía.

Simplifica!!

Como ya comentamos en el Capítulo 15, este modelo de iluminación es una simplificación del modelo físicamente correcto que se resuelve con mejores aproximaciones de la ecuación de Rendering de James Kajiya.

Directamente asociado al concepto de iluminación encontramos las sombras. Gracias a la proyección de sombras, podemos establecer relaciones espaciales entre los objetos de la escena. Por ejemplo, en la Figura 16.1, gracias al uso de sombras podemos saber la posición exacta de la esfera relativa a la escalera.

A continuación estudiaremos los principales tipos de luz que suelen emplearse en videojuegos, así como los modelos de sombreado estáticos y dinámicos más utilizados.

16.2. Tipos de Fuentes de Luz

Las fuentes de luz pueden representarse de diversas formas, dependiendo de las características que queramos simular en la etapa de rendering.

Para especificar la *cantidad* de energía emitida por una fuente de luz, la *radiometría* (ciencia que se encarga de medir la luz) define la *irradiancia* como la cantidad de fotones que pasan por una superficie por segundo.

En videojuegos suelen permitirse tres tipos de fuentes de luz directamente soportadas por el hardware de aceleración gráfico:

- Las **fuentes puntuales** (*point lights*) irradian energía en todas las direcciones a partir de un punto que define su posición en el espacio. Este tipo de fuentes permite variar su posición pero no su dirección. En realidad, las fuentes de luz puntuales no existen como tal en el mundo físico (cualquier fuente de luz tiene asociada un área, por lo que para realizar simulaciones *realistas* de la iluminación tendremos que trabajar con fuentes de *área*. Ogre define este tipo de fuente como `LT_POINT`.
- Uno de los tipos de fuentes más sencillo de simular son las denominadas **fuentes direccionales** (*directional lights*), que pueden considerarse fuentes situadas a una distancia muy grande, por lo que los rayos viajan en una única dirección en la escena (son paralelos entre sí). El *sol* podría ser modelado mediante una fuente de luz direccional. De este modo, la dirección viene determinada por un vector *l* (especificado en coordenadas universales). Este tipo de fuentes de luz no tienen por tanto una posición asociada (únicamente dirección). Este tipo de fuente está descrito como `LT_DIRECTIONAL` en Ogre.

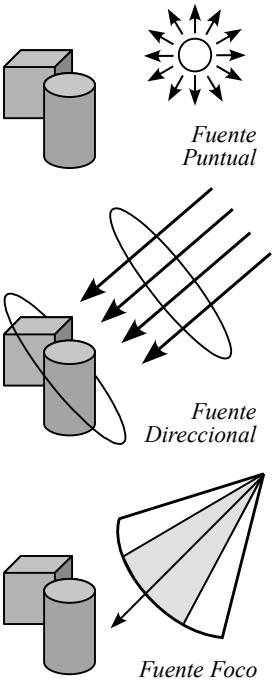


Figura 16.2: Tipos de fuentes de luz utilizadas en aplicaciones interactivas. Ogre soporta únicamente estos tres tipos de fuentes básicos.

- Finalmente los **focos** (*spot lights*) son en cierto modo similares a las fuentes de luz puntuales, pero añadiendo una dirección de emisión. Los focos arrojan luz en forma cónica o piramidal en una dirección específica. De este modo, requieren un parámetro de dirección, además de dos ángulos para definir los conos de emisión interno y externo. Ogre las define como `LT_SPOTLIGHT`.

Las fuentes de luz permiten especificar multitud de parámetros y propiedades, como el color difuso y especular. Una fuente de luz puede definir un color de iluminación difuso (como si el cristal de la *bombilla* estuviera tintado), y un color diferente para el brillo especular. Ambas propiedades están directamente relacionadas con el modo en el que reflejarán la luz los materiales.

En aplicaciones de síntesis de imagen realista suelen definirse además fuentes de luz de área. Este tipo de fuentes simulan el comportamiento de la luz de un modo más realista, donde potencialmente cada punto de la superficie se comporta como un emisor de luz. En la sección 16.7 estudiaremos el modelo de *Radiosidad* que permite trabajar con fuentes de luz de área.

16.3. Sombras Estáticas Vs Dinámicas

La gestión de sombras es un aspecto crítico para dotar de realismo a las escenas sintéticas. Sin embargo, el cálculo de sombras trae asociado un coste computacional importante. De esta forma, en multitud de ocasiones se emplean técnicas de *precálculo* de la iluminación y las sombras. Estos mapas de iluminación permiten generar sombras suaves sin coste computacional adicional.

Ogre soporta dos técnicas básicas de sombreado dinámico: sombreado empleando el *Stencil Buffer* y mediante *Mapas de Texturas* (ver Figura 16.4). Ambas aproximaciones admiten la especificación como *Additive* o *Modulative*.

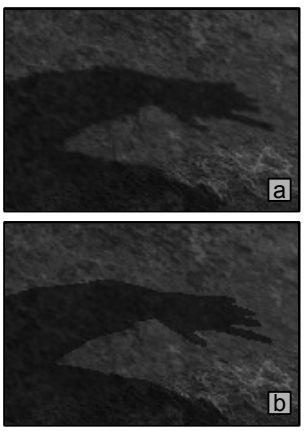


Figura 16.3: Comparativa entre sombras calculadas por **a)** *Mapas de Texturas* y **b)** *Stencil Buffer*. Las basadas en mapas de texturas son claramente dependientes de la resolución del mapa, por lo que deben evitarse con áreas de proyección muy extensas.

En muchas ocasiones se implementan, empleando el Pipeline en GPU programable, algoritmos de sombreado avanzados como aproximaciones al Ambient Occlusion (que veremos en la sección 16.6 o Radiosidad Instantánea (ver sección 16.7).

Las técnicas de tipo *Modulative* únicamente oscurecen las zonas que quedan en sombra, sin importar el número de fuentes de luz de la escena. Por su parte, si la técnica se especifica de tipo *Additive* es necesario realizar el cálculo por cada fuente de luz de modo acumulativo, obteniendo un resultado más preciso (pero más costoso computacionalmente).

Cada técnica tiene sus ventajas e inconvenientes (como por ejemplo, el relativo a la resolución en el caso de los Mapas de Textura, como se muestra en la Figura 16.3). No es posible utilizar varias técnicas a la vez, por lo que deberemos elegir la técnica que mejor se ajuste a las características de la escena que queremos representar. En términos generales, los métodos basados en mapas de textura suelen ser más precisos, pero requieren el uso de tarjetas aceleradoras más potentes. Veamos las características generales de las técnicas de sombreado para estudiar a continuación las características particulares de cada método.

- Únicamente podemos utilizar una técnica de sombreado en la escena. Esta técnica debe ser especificada preferiblemente *antes* de especificar los objetos que formen parte de la escena.
- Las propiedades del material determinan si el objeto arrojará o recibirá sombra. Por defecto los objetos arrojan sombra (salvo los objetos con materiales transparentes, que no arrojarán sombra).
- Es posible definir fuentes de luz que no arrojen sombras.
- En ambos casos es conveniente evitar que las sombras se proyecten de forma extrema (por ejemplo, simulando un amanacer). Este tipo de situaciones hace que la calidad de la sombra se degrade enormemente.
- Dado su coste computacional, por defecto las sombras están *desactivadas* en Ogre.
- Para que un material reciba o arroje sombras, el parámetro *lighting* del material debe estar en *on* (por defecto).

A continuación estudiaremos los detalles de ambos tipos de sombras soportados en Ogre.

16.3.1. Sombras basadas en Stencil Buffer

Empleando esta técnica de sombreado, la forma de la sombra que será proyectada se obtiene proyectando la silueta del objeto calculada desde la perspectiva de la fuente de luz.

El *Stencil Buffer* es un buffer extra disponible en las GPUs modernas que permite almacenar un byte por píxel. Habitualmente se emplea para *recortar* el área de renderizado de una escena. En el cálculo de sombras, se utiliza en combinación con el *ZBuffer* para recortar la zona de sombra relativa al punto de vista. El proceso de cálculo puede resumirse en los siguientes pasos (ver Figura 16.5):

1. Para cada fuente de luz, obtener la lista de aristas de cada objeto que comparten polígonos cuyo vector normal apunta "*hacia*" la fuente de luz y las que están "*opuestas*" a la misma. Estas aristas definen la *silueta* del objeto desde el punto de vista de la fuente de luz.
2. Proyectar estas *aristas de silueta* desde la fuente de luz hacia la escena. Obtener el volumen de sombra proyectado sobre los objetos de la escena.
3. Finalmente, utilizar la información de profundidad de la escena (desde el punto de vista de la cámara virtual) para definir en el *Stencil Buffer* la zona que debe recortarse de la sombra (aquella cuya profundidad desde la cámara sea mayor que la definida en el *ZBuffer*). La Figura 16.5 muestra cómo la proyección del volumen de sombra es recortado para aquellos puntos cuya profundidad es menor que la relativa a la proyección del volumen de sombra.

Esta técnica de generación de sombras, a diferencia de las basadas en Mapas de Textura, utiliza ciclos de la CPU para el renderizado (las operaciones relativas al cálculo de aristas y proyección sobre la escena 3D). A continuación se describen algunas de las características fundamentales de las sombras basadas en Stencil Buffer.

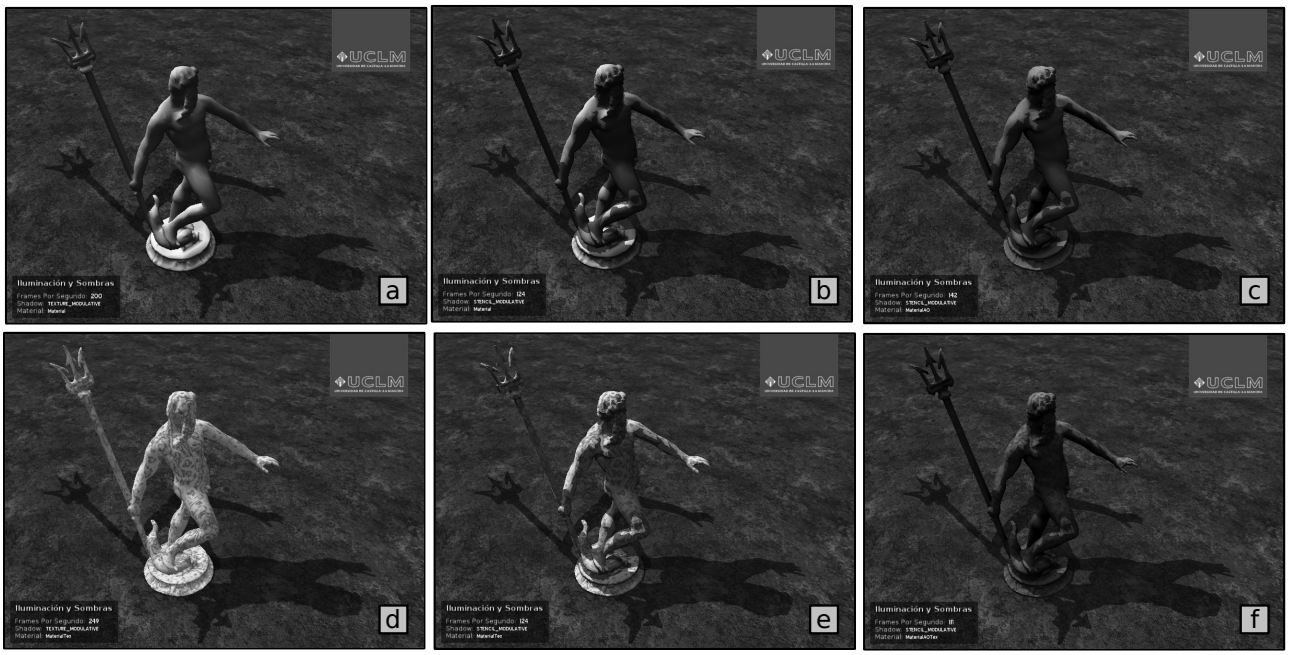


Figura 16.4: Utilización de diversos modos de cálculo de sombras y materiales asociados. **a)** Sombras mediante Mapas de Texturas y Material simple. **b)** Sombras por Stencil Buffer y Material simple. **c)** Sombras por Stencil Buffer y Material con Ambient Occlusion precalculado. **d)** Sombras mediante Mapas de Texturas y Material basado en textura de mármol *procedural* precalculada. **e)** Sombras por Stencil Buffer y Material basado en textura de mármol *procedural* precalculada. **f)** Sombras por Stencil Buffer y Material combinado de c) + e) (Ambient Occlusion y Textura de Mármol precalculados).

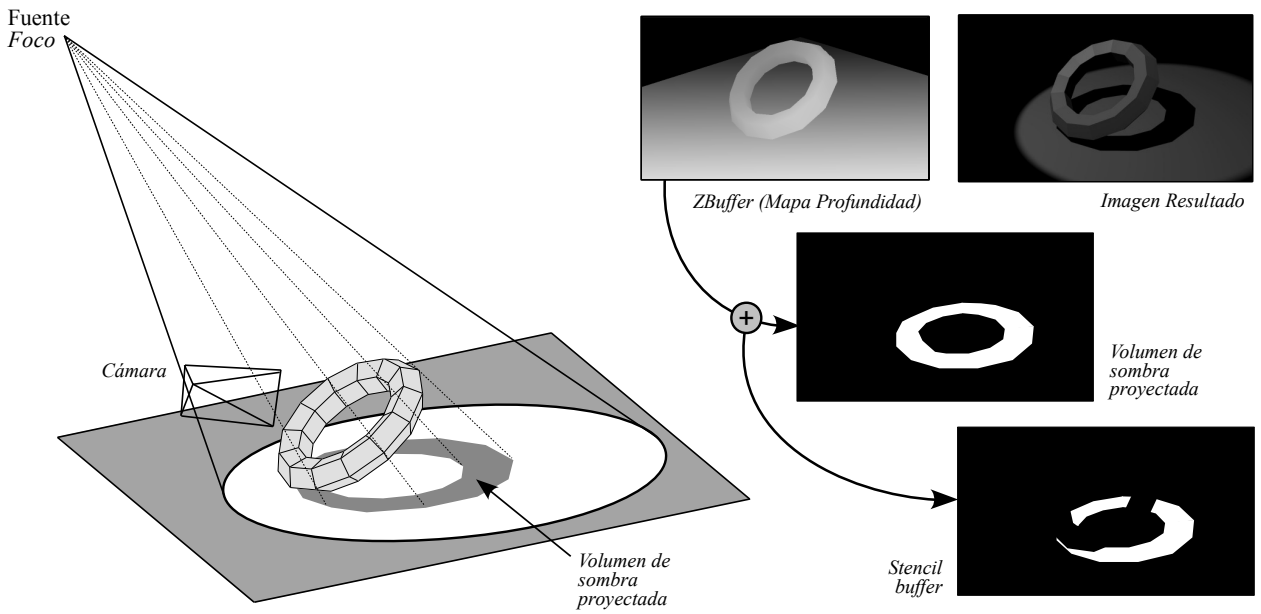


Figura 16.5: Proceso de utilización del *Stencil Buffer* para el recorte de la proyección del volumen de sombra.

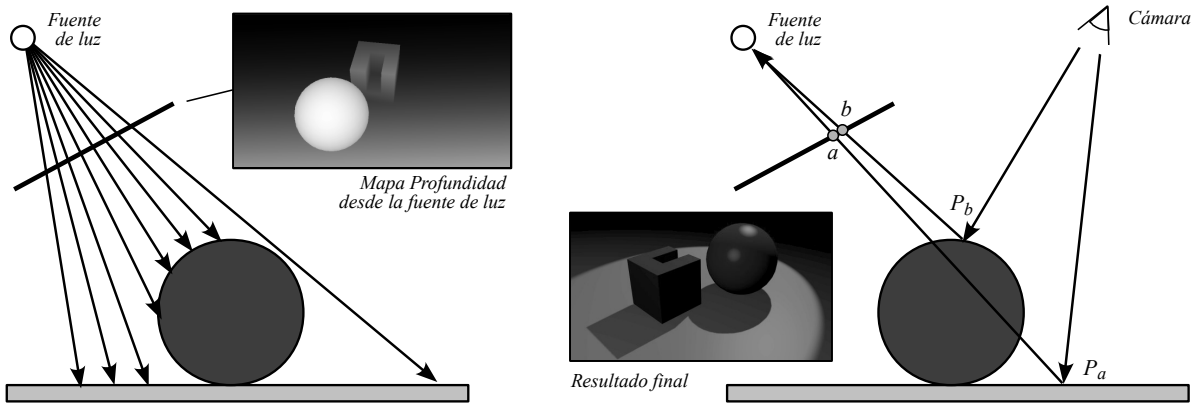


Figura 16.7: Pasos en el cálculo de sombras mediante mapas de textura.

- Empleando el Stencil Buffer, las sombras de objetos transparentes emitirán sombras totalmente sólidas. Es posible desactivar totalmente las sombras de este tipo de objetos, pero empleando Stencil Buffer no es posible obtener sombras semitransparentes.
- Empleando esta técnica no es posible obtener sombras con aristas suaves. En el caso de necesitar este tipo de sombras será necesario emplear la técnica basada de Mapas de Textura (ver Sección 16.3.2).
- Esta técnica permite que el objeto reciba su propia sombra, como se muestra en la Figura 16.6.b.
- Es necesario que Ogre conozca el conjunto de aristas que definen el modelo. Exportando desde Blender, la utilidad *OgreXMLConverter* realiza el cálculo automáticamente. Sin embargo, si implementamos nuestro propio cargador, deberemos llamar a *buildEdgeList* de la clase *Mesh* para que construya la lista.

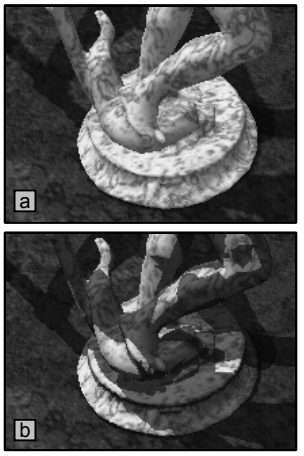


Figura 16.6: En el cálculo de sombras mediante Mapas de Textura (en **a)**), el objeto emisor de sombras no recibe su propia sombra proyectada. Empleando el Stencil Buffer (**b)**) es posible recibir la sombra proyectada.

Z-Fighting

La distinción en grupos de emisores y receptores de sombras se crea para evitar problemas de *Z-fighting* (cuando dos o más elementos tienen asociada la misma profundidad en el ZBuffer y su representación es incorrecta).

16.3.2. Sombras basadas en Texturas

El cálculo de las sombras basadas en texturas se basa en un simple principio: si observamos una escena desde el punto de vista de la fuente de luz, cualquier punto situado *detrás* de lo que ve la fuente de luz estará en sombra. Esta idea puede realizarse en dos pasos principales empleando texturas calculadas directamente en la GPU.

En el primer paso se construye el mapa de profundidad desde el punto de vista. Este mapa simplemente codifica la distancia menor de los objetos de la escena a la fuente de luz (como se muestra en la parte izquierda de la Figura 16.7).

El segundo paso utiliza esta información para construir la sombra. Se calcula la distancia del objeto a la cámara y compara esta distancia con la codificada en el mapa de profundidad. Si la distancia entre cada punto y la fuente de luz es mayor que la almacenada en dicho mapa, el punto está en sombra (Figura 16.7 derecha).

Empleando sombras basadas en texturas el objeto debe ser definido como receptor o emisor de sombras. Un objeto no puede ser emisor y receptor de sombras a la vez (por lo que un emisor no puede recibir sus propias sombras).

A continuación enumeraremos algunas de las características fundamentales de este tipo de técnica.

- Este tipo de sombras permiten el manejo correcto de la transparencia. Además, las sombras pueden tener un color propio.
- Se basan principalmente en el uso de la GPU, por lo que descargan en gran medida la CPU. Directamente relacionado con esta característica, las sombras basadas en mapas de textura permiten componer otros efectos en la GPU (como deformaciones empleando un *Vertex Shader*).
- Esta técnica no permite que el objeto reciba sus propias sombras.

16.4. Ejemplo de uso

El siguiente ejemplo de uso construye una aplicación que permite elegir entre el uso de texturas basadas en *Stencil Buffer* o en *Mapas de Textura*.

Listado 16.1: Fragmento de MyApp.cpp

```
1  void MyApp::createScene() {
2    _sceneManager->setShadowTechnique(SHADOWTYPE_STENCIL_MODULATIVE);

3    _sceneManager->setShadowColour(ColourValue(0.5, 0.5, 0.5));
4    _sceneManager->setAmbientLight(ColourValue(0.9, 0.9, 0.9));
5
6    _sceneManager->setShadowTextureCount(2);
7    _sceneManager->setShadowTextureSize(512);
8
9    Light* light = _sceneManager->createLight("Light1");
10   light->setPosition(-5,12,2);
11   light->setType(Light::LT_SPOTLIGHT);
12   light->setDirection(Vector3(1,-1,0));
13   light->setSpotlightInnerAngle(Degree(25.0f));
14   light->setSpotlightOuterAngle(Degree(60.0f));
15   light->setSpotlightFalloff(0.0f);
16   light->setCastShadows(true);
17
18   Light* light2 = _sceneManager->createLight("Light2");
19   light2->setPosition(3,12,3);
20   light2->setDiffuseColour(0.2,0.2,0.2);
21   light2->setType(Light::LT_SPOTLIGHT);
22   light2->setDirection(Vector3(-0.3,-1,0));
23   light2->setSpotlightInnerAngle(Degree(25.0f));
24   light2->setSpotlightOuterAngle(Degree(60.0f));
25   light2->setSpotlightFalloff(5.0f);
26   light2->setCastShadows(true);
27
28   Entity* ent1 = _sceneManager->createEntity("Neptuno.mesh");
29   SceneNode* node1 = _sceneManager->createSceneNode("Neptuno");
30   ent1->setCastShadows(true);
31   node1->attachObject(ent1);
32   _sceneManager->getRootSceneNode()->addChild(node1);
33
34   // ... Creamos plano1 manualmente (codigo eliminado)
35
36   SceneNode* node2 = _sceneManager->createSceneNode("ground");
37   Entity* groundEnt = _sceneManager->createEntity("p", "plane1");
38   groundEnt->setMaterialName("Ground");
39   groundEnt->setCastShadows(false);
40   node2->attachObject(groundEnt);
41   _sceneManager->getRootSceneNode()->addChild(node2);
42 }
```

En la línea [2] se define la técnica de cálculo de sombras que se utilizará por defecto, aunque en el *FrameListener* se cambiará en tiempo de ejecución empleando las teclas [1] y [2]. Las líneas [2-3] definen propiedades generales de la escena, como el color con el que se representarán las sombras y el color de la luz ambiental (que en este ejemplo, debido a la definición de los materiales, tendrá especial importancia).

Las líneas [6-9] únicamente tienen relevancia en el caso del uso del método de cálculo basado en mapas de textura. Si el método utilizado es el de *Stencil Buffer* simplemente serán ignoradas. La línea [6] configura el número de texturas que se emplearán para calcular las sombras. Si se dispone de más de una fuente de luz (como en este ejemplo), habrá que especificar el número de texturas a utilizar.

El tamaño de la textura (cuadrada) se indica en la línea [7]. A mayor tamaño, mejor resolución en la sombra (pero mayor cantidad de memoria gastada). El tamaño por defecto es 512, y *debe ser potencia de dos*.

A continuación en las líneas [9-26] se definen dos fuente de luz de tipo *SPOTLIGHT*. Cada luz define su posición y rotación. En el caso de la segunda fuente de luz se define además un color difuso (línea [20]). El ángulo interno y externo [13-14] define el cono de iluminación de la fuente. En ambas fuentes se ha activado la propiedad de que la fuente de luz permita el cálculo de sombras.

El método de *setCastShadows* pertenece a la clase *MovableObject*. En el caso de entidades, si se especifica como cierto, el objeto arrojará sombras sobre otros objetos. En caso contrario, funcionará como un receptor de sombras. El mismo método se aplica sobre luces (que, es un tipo específico de *MovableObject*). En el caso de fuentes de luz, el mismo método sirve para especificar si esa fuente de luz se utilizará en el cálculo de sombras.

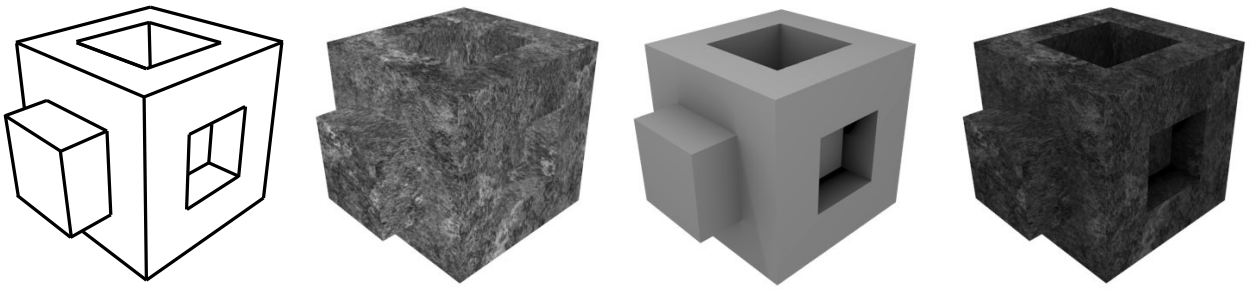
Como hemos comentado en la sección 16.3.2, si utilizamos mapas de textura para calcular sombras, un objeto puede funcionar únicamente como receptor o como emisor de sombras. En este ejemplo, el plano se configura como receptor de sombras (en la línea [39]), mientras que el objeto *Neptuno* funciona como emisor (línea [30]).

Mediante las teclas [7] ... [0] es posible cambiar el material asociado al objeto principal de la escena. En el caso del último material definido para el objeto (tecla [0]), se compone en dos capas de textura el color base (calculado mediante una textura procedural) y una capa de iluminación basada en Ambient Occlusion. La definición del material compuesto se muestra en la Figura 16.8. Gracias a la separación de la iluminación y del color en diferentes mapas, es posible cambiar la resolución individualmente de cada uno de ellos, o reutilizar el mapa de iluminación en diferentes modelos que compartan el mismo mapa de iluminación pero tengan diferente mapa de color.

Multitud de juegos utilizan el precálculo de la iluminación. Quake II y III utilizaron modelos de Radiosidad para crear mapas de iluminación para simular de una forma mucho más realista el comportamiento físico de la luz en la escena. Aunque actualmente poco a poco se implantan las técnicas de iluminación dinámicas a nivel de píxel, los mapas de iluminación siguen siendo una opción ampliamente utilizada.

```
material MaterialAOTex
{
 receive_shadows on
 technique
 {
   pass
   {
     texture_unit
     {
     texture
       neptuno_tex.jpg
     }
     texture_unit
     {
     texture
       neptuno_lm.jpg
       colour_op_ex
         modulate
         src_texture
         src_current
     }
    }
  }
}
```

Figura 16.8: Definición del material multicapa para componer una textura con iluminación de tipo AbmientOcclusion con la textura de color. Ambas texturas emplean un despliegue automático (tipo *LightMap* en Blender).



Geometría base | *Mapa de color* | *Mapa de Iluminación (Ambient Occlusion)* | *Resultado Final*

Figura 16.9: Ejemplo de uso de un mapa de iluminación para definir sombras suaves en un objeto. El mapa de color y el mapa de iluminación son texturas independiente que se combinan (multiplicando su valor) en la etapa final.

16.5. Mapas de Iluminación

Como hemos visto anteriormente, los modelos de iluminación local calculan en cada vértice del modelo la interacción con la luz. El color final de cada punto del modelo se calcula mediante técnicas de interpolación. Las sombras obtenidas empleando estos métodos no son precisas (por ejemplo, no se pueden calcular sombras difusas).

En los últimos años las tarjetas gráficas permiten el cálculo de métodos de iluminación por píxel (*per pixel lighting*), de modo que por cada píxel de la escena se calcula la contribución *real* de la iluminación. El cálculo preciso de las sombras es posible, aunque a día de hoy todavía es muy costoso para videojuegos.

Los mapas de luz permiten precalcular la iluminación por píxel de forma estática. Esta iluminación precalculada puede ser combinada sin ningún problema con los métodos de iluminación dinámicos estudiados hasta el momento. La calidad final de la iluminación es únicamente dependiente del tiempo invertido en el precálculo de la misma y la resolución de las texturas.

En términos generales, el píxel de una textura se denomina *texel* (de *Texture Element*). De forma análoga, un píxel de un mapa de iluminación se denomina *lumel* (*Lumination Element*).

De este modo, para cada cara poligonal del modelo se definen una o varias capas de textura. La capa del mapa de iluminación es finalmente *multiplicada* con el resultado de las capas anteriores (como se puede ver en la Figura 16.9).

A diferencia de las texturas de color donde cada vértice del modelo puede compartir las mismas coordenadas UV con otros vértices, en mapas de iluminación cada vértice de cada cara *debe* tener una coordenada única. Los mapas de iluminación se cargan de la misma forma que el resto de texturas de la escena. En la Figura 16.8 hemos estudiado un modo sencillo de composición de estos mapas de iluminación,

aunque pueden definirse otros métodos que utilicen varias pasadas y realicen otros modos de composición. A continuación estudiaremos dos técnicas ampliamente utilizadas en el precálculo de la iluminación empleando mapas de iluminación: Ambient Occlusion y Radiosidad.

Aunque nos centremos en los métodos de Ambient Occlusion y Radiosidad, Blender permite precalcular (*Render Baking*) cualquier configuración de escena. De este modo, se puede precalcular cualquier configuración de luces y materiales de la escena.

Como hemos comentado antes, cualquiera de los métodos de cálculo de mapas de iluminación requiere que cada cara poligional del modelo tenga unas coordenadas UV únicas en el modelo. Como estudiamos en la sección 15.5, empleando el despliegue de tipo *Lightmap Pack* conseguimos que Blender despliegue el modelo de esta forma, asignando el área de la imagen de forma proporcional al tamaño de cada cara del modelo. La Figura 16.10 muestra el despliegue realizado para el modelo de Neptuno tras aplicar Ambient Occlusion.

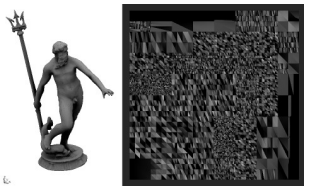


Figura 16.10: Despliegue del modelo de Neptuno empleando *Lightmap Pack*. Cada cara poligonal tiene asociado un espacio propio en la textura.

Una vez realizado el despliegue y con una imagen asociada al despiegue UV (puede crearse una imagen *nueva* vacía de color sólido desde la cabecera de la ventana UV Image Editor en el menú *Image/ New Image*), el precálculo de la iluminación se realiza en la pestaña *Bake* del grupo de botones de Render (ver Figura 16.11).

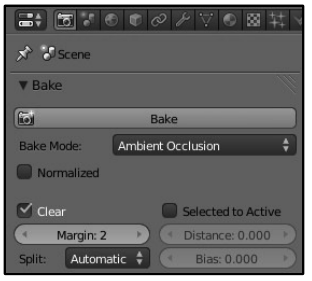


Figura 16.11: Opciones de la pestaña Bake del grupo de botones de Render.

A continuación se estudiarán las opciones más relevantes en el ámbito del precálculo de la iluminación.

Mediante la lista desplegable *Bake Mode*, se elige el tipo de precálculo que va a realizarse. Con *Full Render* se realizará el cálculo de todas las propiedades del material, texturas e iluminación sobre la textura (sin tener en cuenta el brillo especular que es dependiente del punto de vista del observador). Si se elige la opción *Ambient Occlusion* (como se muestra en la figura 16.11), únicamente se tendrá en cuenta la información de AO ignorando el resto de fuentes de luz de la escena (ver sección 16.6). Mediante la opción *Textures* se asignan los colores base de los materiales y texturas (sin tener en cuenta el sombreado).

El checkbox *Clear* sirve para borrar la textura antes de realizar el *baking*. Puede ser interesante desactivarlo si queremos aplicar una pasada de AO a una textura previamente asignada manualmente.

El botón *Selected to Active* permite asignar la información de otro objeto. Un uso típico de esta herramienta es disponer de un modelo en alta resolución que define una geometría muy detallada, y mapear su información en otro modelo de baja resolución. En esta sección utilizaremos esta funcionalidad para asociar la información de una malla de radiosidad a la malla en baja resolución de la escena.

Finalmente, cuando se han elegido los parámetros en la pestaña, pulsando el botón Bake se inicia el proceso de cálculo. En la ventana de UV Mapping deberá verse la actualización en tiempo real de la textura mapeada al modelo.

16.6. Ambient Occlusion

El empleo del término de luz ambiente viene aplicándose desde el inicio de los gráficos por computador como un método muy rápido de simular la contribución de luz ambiental que proviene de todas las direcciones. La técnica de *Ambient Occlusion* es un caso particular del uso de pruebas de oclusión en entornos con iluminación local para determinar los efectos difusos de iluminación. Estas técnicas fueron introducidas inicialmente por Zhurov como alternativa a las técnicas de radiosidad para aplicaciones interactivas (videojuegos), por su bajo coste computacional.

En el esquema de la figura 16.12 podemos ver en qué se basan estas técnicas. Desde cada punto P de intersección con cada superficie (obtenido mediante trazado de rayos), calculamos el valor de ocultación de ese punto que será proporcional al número de rayos que alcanzan el "*cielo*" (los que no intersectan con ningún objeto dada una distancia máxima de intersección). En el caso de la figura serán $4/7$ de los rayos lanzados.

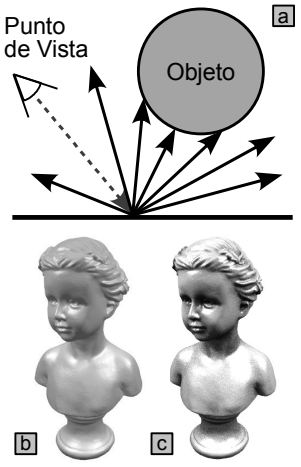


Figura 16.12: Descripción esquemática del cálculo de Ambient Occlusion. **a)** Esquema de cálculo del valor de ocultación $W(P)$. **b)** Render obtenido con iluminación lobal (dos fuentes puntuales). **c)** La misma configuración que en b) pero con Ambient Occlusion de 10 muestras por píxel.

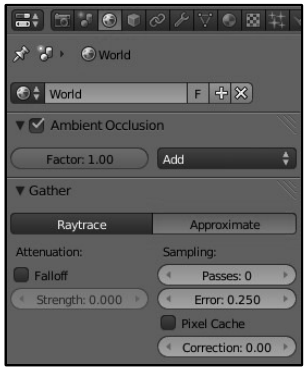


Figura 16.13: Pestaña Amb Occ en Blender.

Podemos definir la ocultación de un punto de una superficie como:

$$W(P) = \frac{1}{\pi} \int_{\omega \in \Omega} \rho(d(P,\omega)) cos\theta d\omega \tag{16.1}$$

Obteniendo un valor de ocultación $W(P)$ entre 0 y 1, siendo $d(P,\omega)$ la distancia entre P y la primera intersección con algún objeto en la dirección de ω. $\rho(d(P,\omega))$ es una función con valores entre 0 y 1 que nos indica la magnitud de iluminación ambiental que viene en la dirección de ω, y θ es el ángulo formado entre la normal en el punto P y la dirección de ω.

Estas técnicas de ocultación (*obscurances*) se desacoplan totalmente de la fase de iluminación local, teniendo lugar en una segunda fase de iluminación secundaria difusa. Se emplea un valor de distancia para limitar la ocultación únicamente a polígonos cercanos a la zona a sombrear mediante una función. Si la función toma valor de ocultación igual a cero en aquellos puntos que no superan un umbral y un valor de uno si están por encima del umbral, la técnica de ocultación se denomina *Ambient Occlusion*. Existen multitud de funciones exponenciales que se utilizan para lograr estos efectos.

La principal ventaja de esta técnica es que es bastante más rápida que las técnicas que realizan un cálculo correcto de la iluminación indirecta. Además, debido a la sencillez de los cálculos, pueden realizarse aproximaciones muy rápidas empleando la GPU, pudiendo utilizarse en aplicaciones interactivas. El principal inconveniente es que no es un método de iluminación global y no puede simular efectos complejos como caústicas o contribuciones de luz entre superficies con reflexión difusa.

Para activar el uso de Ambient Occlusion (AO) en Blender, en el grupo de botones del mundo , en la pestaña *Ambient Occlusion* activamos el checkbox (ver Figura 16.13). La pestaña *Gather* permite elegir el tipo de recolección de muestras entre trazado de rayos *Raytrace* o Aproximada *Approximate* (ver Figura 16.13).

El cálculo de AO mediante **Trazado de Rayos** ofrece resultados más precisos a costa de un tiempo de render mucho mayor. El efecto del ruido blanco debido al número de rayos por píxel puede disminuirse a costa de aumentar el tiempo de cómputo aumentando el número de muestras (Samples).

El método AO **Aproximado** realiza una rápida aproximación que, en muchos casos, puede ser suficiente. No sufre de ruido de muestreo, por lo que es buena opción para ser utilizada en multitud de ocasiones. El parámetro *Error* define la calidad de las sombras calculadas (valores menores implican mejores resultados con mayor tiempo de cómputo). El checkbox *Pixel Cache* si está activo hace que el valor de sombreado se interpole entre píxeles vecinos, haciendo que el cálculo sea aún más rápido (aunque menos exacto). A continuación estudiaremos algunas opciones relevantes del método:

- **Falloff**: Esta opción controla el tamaño de las sombras calculadas por AO. Si está activa, aparece un nuevo control *Strength* que permite variar el factor de atenuación. Con valores mayores de *Strength*, la sombra aparece más enfocada (es más pequeña).
- **Add** (Por defecto): El punto recibe luz según los rayos que no se han chocado con ningún objeto. La escena esta más luminosa que la original sin AO.
- **Multiply**: El punto recibe sombra según los rayos que han chocado con algún objeto. La escena es más oscura que la original sin AO.

16.7. Radiosidad

En esta técnica se calcula el intercambio de luz entre superficies. Esto se consigue subdividiendo el modelo en pequeñas unidades denominadas *parches*, que serán la base de la distribución de luz final. Inicialmente los modelos de radiosidad calculaban las interacciones de luz entre superficies difusas (aquellas que reflejan la luz igual en todas las direcciones), aunque existen modelos más avanzados que tienen en cuenta modelos de reflexión más complejos.

El modelo básico de radiosidad calcula una solución independiente del punto de vista. Sin embargo, el cálculo de la solución es muy costoso en tiempo y en espacio de almacenamiento. No obstante, cuando la iluminación ha sido calculada, puede utilizarse para renderizar la escena desde diferentes ángulos, lo que hace que este tipo de soluciones se utilicen en visitas interactivas y videojuegos en primera persona actuales. En el ejemplo de la figura 16.15, en el techo de la habitación se encuentran las caras poligonales con propiedades de emisión de luz. Tras aplicar el proceso del cálculo de radiosidad obtenemos la malla de radiosidad que contiene información sobre la distribución de iluminación entre superficies difusas.

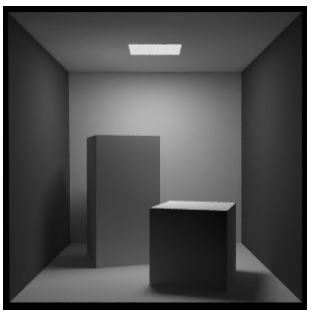


Figura 16.14: El modelo de Radiosidad permite calcular el intercambio de luz entre superficies difusas, mostrando un resultado de render correcto en el problema planteado en la histórica *Cornell Box*.

En el modelo de radiosidad, cada superficie tiene asociados dos valores: la intensidad luminosa que recibe, y la cantidad de energía que emite (energía radiante). En este algoritmo se calcula la interacción de energía desde cada superficie hacia el resto. Si tenemos n superficies, la complejidad del algoritmo será $O(n^2)$. El valor matemático que calcula la relación geométrica entre superficies se denomina **Factor de Forma**, y se define como:

$$F_{ij} = \frac{cos\theta_i cos\theta_j}{\pi r^2} H_{ij} dA_j \tag{16.2}$$

Siendo F_{ij} el factor de forma de la superficie i a la superficie j, en el numerador de la fracción definimos el ángulo que forman las normales de las superficies, πr^2 mide la distancia entre las superficies, H_{ij} es el parámetro de visibilidad, que valdrá uno si la superfice j es totalmente visible desde i, cero si no es visible y un valor entre uno y cero según el nivel de oclusión. Finalmente, dA_j indica el área de la superfice j (no tendremos el mismo resultado con una pequeña superficie emisora de luz sobre una superfice grande que al contrario). Este factor de forma se suele calcular empleando un hemi-cubo.

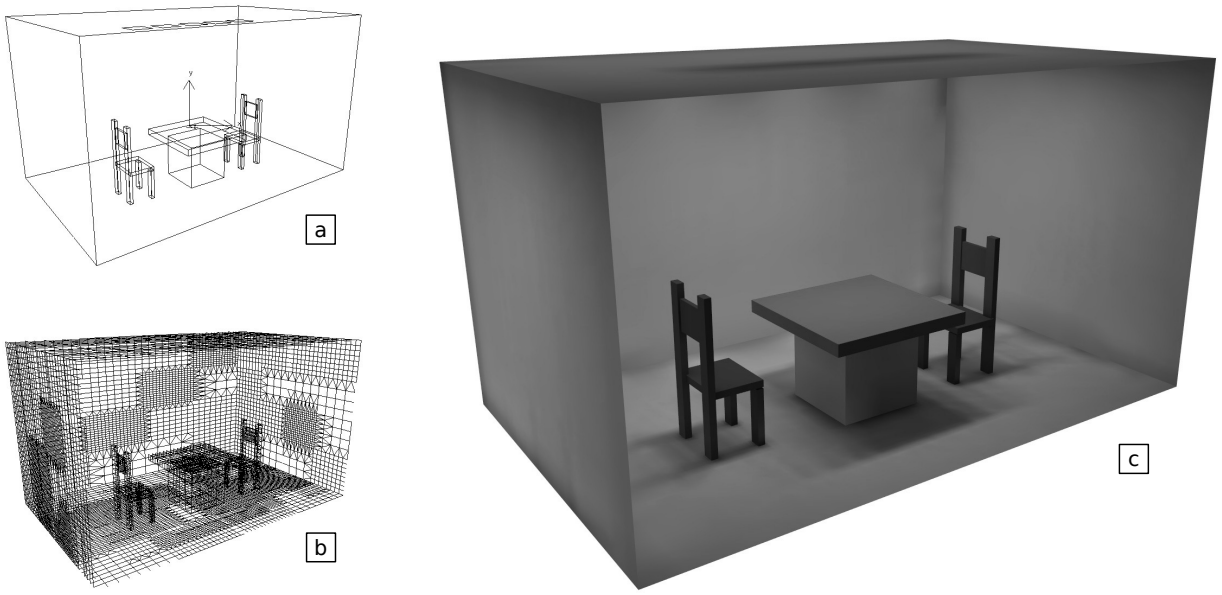


Figura 16.15: Ejemplo de aplicación del método de radiosidad sobre una escena simple. En el techo de la habitación se ha definido una fuente de luz. **a)** Malla original. **b)** Malla de radiosidad. **c)** Resultado del proceso de renderizado.

Será necesario calcular n^2 factores de forma (no cumple la propiedad conmutativa) debido a que se tiene en cuenta la relación de área entre superficies. La matriz que contiene los factores de forma relacionando todas las superficies se denomina *Matriz de Radiosidad*. Cada elemento de esta matriz contiene un factor de forma para la interacción desde la superficie indexada por la columna hacia la superficie indexada por la fila (ver figura 16.16).

```
Para cada iteración
  seleccionar un parche i
  calcular Fij para todas las superficies j
  para cada superficie j hacer:
    actualizar la radiosidad de la superficie j
    actualizar la emisión de la superficie j
  emision(i) = 0
```

En 1988, Cohen introdujo una variante de cálculo basada en el *refinamiento progresivo* que permite que la solución de radiosidad encontrada en cada iteración del algoritmo sea mostrada al usuario. Este método progresivo es un método incremental que requiere menos tiempo de cómputo y de almacenamiento; en cada iteración se calcula los factores de forma entre una superficie y el resto (en el artículo original se requería el cálculo de n^2 factores de forma).

Este método de refinamiento progresivo finalmente obtiene la misma solución que el original, proporcionando resultados intermedios que van siendo refinados.

Refinamiento Progresivo

Blender implementa el método de Radiosidad basado en refinamiento progresivo

En general, el cálculo de la radiosidad es eficiente para el cálculo de distribuciones de luz en modelos simples con materiales difusos, pero resulta muy costoso para modelos complejos (debido a que se calculan los valores de energía para cada parche del modelo) o con materiales no difusos. Además, la solución del algoritmo se muestra

como una nueva malla poligonal que tiende a desenfocar los límites de las sombras. Como en videojuegos suelen emplearse modelos en baja poligonalización, y es posible *mapear* esta iluminación precalculada en texturas, el modelo de Radiosidad se ha empleado en diveros títulos de gran éxito comercial. Actualmente incluso existen motores gráficos que calculan soluciones de radiosidad en tiempo real.

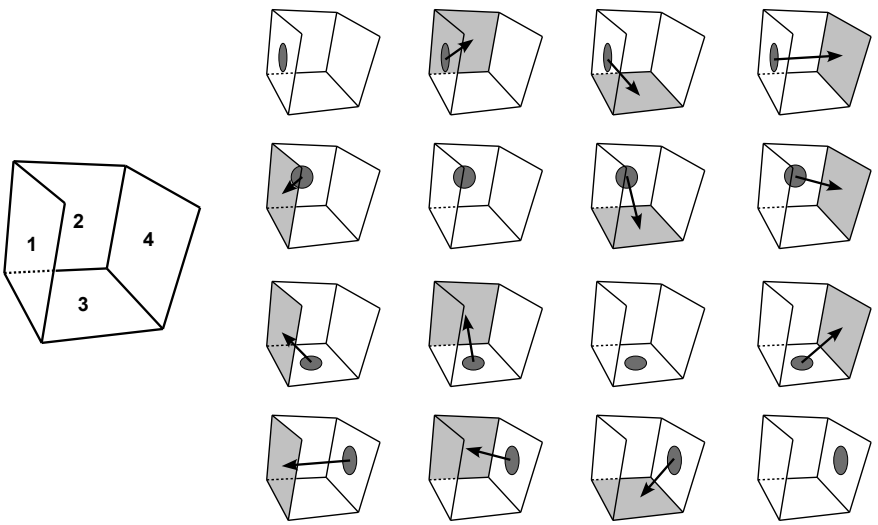


Figura 16.16: Esquema de la matriz de radiosidad. En cada posición de la matriz se calcula el *Factor de Forma*. La diagonal principal de la matriz no se calcula.

Veamos a continuación un ejemplo de utilización de Radiosidad en Blender. Como hemos indicado anteriormente, el modelo de Radiosidad calcula la interacción de iluminación entre superficies. Así, la iluminación de la escena vendrá dada por las áreas de luz de las superficies superiores (planos) de la habitación. De esta forma, necesitaremos crear planos a los que asignaremos un material que emita luz. La escena de ejemplo (ver Figura 16.17) cuenta con planos con estas propiedades definidas.

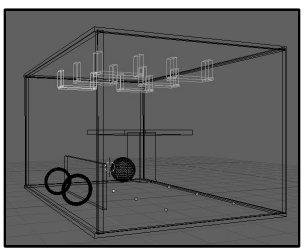


Figura 16.17: Escena de ejemplo para el cálculo de la Radiosidad.

Necesario Blender 2.49b. Aunque la radiosidad es un método muy empleado en videojuegos, fue eliminado de la implementación de Blender a partir de la versión 2.5. Probablemente vuelva a incorporarse en futuras versiones, pero de momento es necesario utilizar versiones anteriores del programa. La última versión que la incorporó (Blender 2.49b) puede ser descargada de la página oficial http://download.blender.org/release/.

Es muy importante la dirección del vector normal ya que Blender calculará la interacción de la luz en ese sentido. Por tanto, tendremos que asegurarnos que el vector normal de cada foco apunta "*hacia el suelo*", tal y como muestra la Figura 16.18. Para comprobar que es así, seleccionamos cada emisor de luz, en modo de edición de vértices activamos el botón *Draw Normals* de la pestaña *Mesh Tools More*. Podemos ajustar el tamaño de representación del vector normal en *Nsize*. En caso de que la normal esté invertida, podemos ajustarla pulsando [W] *Flip Normals*.

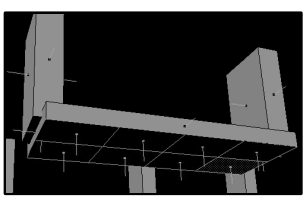


Figura 16.18: Focos del techo con el vector normal correctamente definido.

Figura 16.19: Opciones generales de Radiosidad (Paneles *Radio Render*, *Radio Tool* y *Calculation*.

Los elementos que emiten luz tendrán el campo *Emit* del material con un valor mayor que 0. Los emisores de este ejemplo tienen un valor de emisión de 0.4.

Accedemos al menú de radiosidad dentro de los botones de sombreado . Seleccionamos todas las mallas que forman nuestra escena [A] y pinchamos en el botón *Collect Meshes* (ver Figura 16.19). La escena aparecerá con colores sólidos.

Hecho esto, pinchamos en [Go]. De esta forma comienza el proceso de cálculo de la solución de radiosidad. Podemos parar el proceso en cualquier momento pulsando [Escape], quedándonos con la aproximación que se ha conseguido hasta ese instante.

Si no estamos satisfechos con la solución de Radiosidad calculada, podemos eliminarla pinchando en *Free Radio Data*.

En la Figura 16.19 se muestran algunas opciones relativas al cálculo de la solución de radiosidad. El tamaño de los *Parches* (*PaMax* - *PaMin*) determina el detalle final (cuanto más pequeño sea, más detallado será el resultado final), pero incrementamos el tiempo de cálculo. De forma similar ocurre con el tamaño de los *Elementos*[2] (*ElMax* - *ElMin*). En *Max Iterations* indicamos el número de pasadas que Blender hará en el bucle de Radiosidad. Un valor 0 indica que haga las que estime necesarias para minimizar el error (lo que es conveniente si queremos generar la malla de radiosidad final). *MaxEl* indica el número máximo de elementos para la escena. Hemires es el tamaño del hemicubo para el cálculo del factor de forma.

Una vez terminado el proceso de cálculo (podemos pararlo cuando la calidad sea aceptable con [Esc]), podemos añadir la nueva malla calculada reemplazando las creadas anteriormente (*Replace Meshes*) o añadirla como nueva a la escena (*Add new Meshes*). Elegiremos esta segunda opción, para aplicar posteriormente el resultado a la malla en baja poligonalización. Antes de deseleccionar la malla con la información de radiosidad calculada, la moveremos a otra capa (mediante la tecla [M]) para tener los objetos mejor organizados. Hecho esto, podemos liberar la memoria ocupada por estos datos pinchando en *Free Radio Data*.

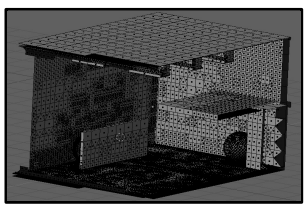


Figura 16.20: Resultado de la malla de radiosidad.

El resultado de la malla de radiosidad se muestra en la Figura 16.20. Como se puede comprobar, es una malla extremadamente densa, que no es directamente utilizable en aplicaciones de tiempo real. Vamos a utilizar el *Baking* de Blender para utilizar esta información de texturas sobre la escena original en baja poligonalización.

Por simplicidad, uniremos todos los objetos de la escena original en una única malla (seleccionándolos todos [A] y pulsando [Control] [J] *Join Selected Meshes*). A continuación, desplegaremos el objeto empleando el modo de despliegue de *Light Map*, y crearemos una nueva imagen que recibirá el resultado del render Baking.

[2]En Blender, cada Parche está formado por un conjunto de Elementos, que describen superficies de intercambio de mayor nivel de detalle.

Figura 16.21: Resultado de aplicación del mapa de radiosidad al modelo en baja resolución.

Ahora seleccionamos primero la malla de radiosidad, y después con Shift pulsado seleccionamos la malla en baja resolución. Pinchamos en el botón *Selected to Active* de la pestaña *Bake*, activamos *Full Render* y pinchamos en el botón Bake. Con esto debemos haber asignado el mapa de iluminación de la malla de radiosidad al objeto en baja poligonalización (ver Figura 16.21).

Capítulo 17

Exportación y Uso de Datos de Intercambio

Carlos González Morcillo

Hasta ahora hemos empleado exportadores genéricos de contenido de animación, mallas poligonales y definición de texturas. Estas herramientas facilitan enormemente la tarea de construcción de contenido genérico para nuestros videojuegos. En este capítulo veremos cómo crear nuestros propios scripts de exportación y su posterior utilización en una sencilla demo de ejemplo.

17.1. Introducción

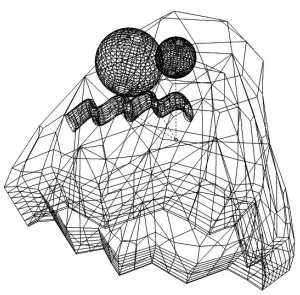


Figura 17.1: El *Wumpus* será el personaje principal del videojuego demostrador de este capítulo *NoEscape Demo*.

En capítulos anteriores hemos utilizado exportadores genéricos de contenido para su posterior importación. En multitud de ocasiones en necesario desarrollar herramientas de exportación de elementos desde las herramientas de diseño 3D a formatos de intercambio.

La comunidad de Ogre ha desarrollado algunos exportadores de geometría y animaciones para las principales suites de animación y modelado 3D. Sin embargo, estos elementos almacenan su posición relativa a su sistema de coordenadas, y no tienen en cuenta otros elementos (como cámaras, fuentes de luz, etc...).

En esta sección utilizaremos las clases definidas en el Capítulo 6 del Módulo 1 para desarrollar un potencial juego llamado *NoEscapeDemo*. Estas clases importan contenido definido en formato XML, que fue descrito en dicho capítulo.

A continuación enumeraremos las características esenciales que debe soportar el exportador a desarrollar:

- **Múltiples nodos**. El mapa del escenario (ver Figura 17.4) describe un grafo. Utilizaremos un objeto de tipo malla poligonal para modelarlo en Blender. Cada vértice de la malla representará un nodo del grafo. Los nodos del grafo podrán ser de tres tipos: *Nodo productor* de Wumpus (tipo **spawn**), *Nodo destructor* de Wumpus (tipo **drain**) o un *Nodo genérico* del grafo (que servirá para modelar las intersecciones donde el Wumpus puede cambiar de dirección).
- **Múltiples aristas**. Cada nodo del grafo estará conectado por uno o más nodos, definiendo múltiples aristas. Cada arista conectará una pareja de nodos. En este grafo, las aristas no tienen asociada ninguna dirección.
- **Múltiples cámaras animadas**. El exportador almacenará igualmente las animaciones definidas en las cámaras de la escena. En el caso de querer almacenar cámaras estáticas, se exportará únicamente un fotograma de la animación. Para cada frame de la animación de cada cámara, el exportador almacenará la posición y rotación (mediante un *cuaternio*) de la cámara en el XML.

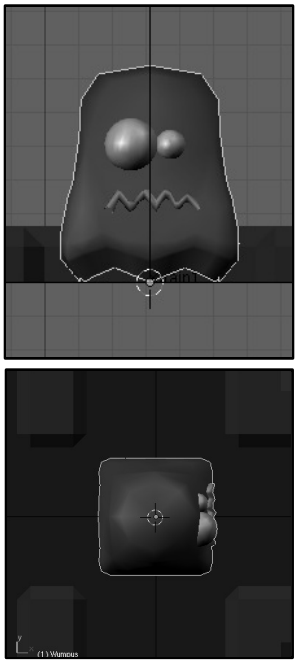


Figura 17.2: Como se estudió en el Capítulo 11, es necesario definir el centro de los objetos especificando adecuadamente su centro. En el caso del Wumpus, el centro se define en el centro de la base, escalándolo además al tamaño existente entre las paredes del escenario.

Existen algunas especificaciones en XML genéricas para la definición de escenas (como el formato "*.scene*") en Ogre, que está soportado por unas clases adicionales disponibles en el Wiki (`http://www.ogre3d.org/tikiwiki/DotScene`) del framework. Aunque este formato define gran cantidad de elementos de una escena, resulta crítico conocer los mecanismos de exportación para desarrollar los scripts específicos que necesitemos en cada proyecto.

El mapa del escenario y los personajes principales se exportarán atendiendo a las indicaciones estudiadas en el Capítulo 11 (ver Figura 17.2). La exportación del XML se realizará empleando el script descrito a continuación. Blender define una API de programación muy completa en Python.

Python es un lenguaje ampliamente utilizado por su facilidad, versatilidad y potencia [1]. Python es un lenguaje de muy alto nivel, interpretado (compilado a un código intermedio), orientado a objetos y libre bajo licencia GPL. Aunque en esta sección nos centraremos en su uso directo como lenguaje de acceso a la API de Blender, en el módulo 3 estudiaremos cómo utilizar Python como lenguaje de script dentro de nuestra apliación en C++.

¿Quién usa Python? Existen multitud de títulos comerciales y videojuegos de éxito que han utilizado Python como lenguaje para su desarrollo. Por ejemplo, *Civilization 4* utiliza python para la gestión de múltiples tareas, *Vampire: The Masquerade* lo emplea para la descripción de Mods, o *Battlefield 2* (de *Digital Illusions*) para la definición de todos los complementos del juego.

La versión de Blender 2.65 utiliza Python 3.3, por lo que será necesaria su instalación para ejecutar los scripts desarrollados. En una ventana de tipo *Text Editor* podemos editar los scripts. Para su ejecución, bastará con situar el puntero del ratón dentro de la ventana de texto y pulsar Alt P.

[1] Resulta especialmente interesante la comparativa empírica de Python con otros lenguajes de programación disponible en `http://www.ipd.uka.de/~{}prechelt/\discretionary{-}{}{}Biblio/jccpprtTR.pdf`

Aunque la versión 2.65 de Blender permite añadir nuevas propiedades a los objetos de la escena, por mantener la compatibilidad con versiones anteriores lo haremos codificándolo como parte del nombre. En la pestaña *Object*, es posible especificar el nombre del objeto, como se muestra en la Figura 17.3. En nuestro ejemplo, se ha utilizado el siguiente convenio de nombrado:

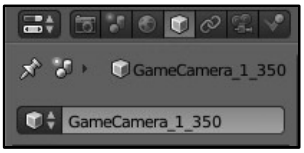


Figura 17.3: Especificación del nombre del objeto en la pestaña *Object*.

- **Cámaras**. Las cámaras codifican en el nombre (separadas por guión bajo), como primer parámetro el identificador de la misma (un índice único), y como segundo parámetro el número de frames asociado a su animación. De este modo, la cámara llamada *GameCamera_1_350* indica que es la primera cámara (que será la utilizada en el estado de *Juego*), y que tiene una animación definida de 350 frames.
- **Empty**. Como no es posible asignar tipos a los vértices de la malla poligonal con la que se definirá el grafo, se han añadido objetos de tipo Empty que servirán para asociar los generadores (*spawn*) y destructores (*drain*) de *Wumpus*. Estos objetos Empty tendrán como nombre el literal "spawn" o "drain" y a continuación un número del 1 al 9.

A continuación se muestra el código fuente del script de exportación. Las líneas 8-10 definen tres constantes que pueden ser modificadas en su ejecución: 8 el nombre del fichero XML que se exportará, 9 el nombre del objeto donde se define el grafo (como hemos indicado anteriormente será una malla poligonal), y 10 un valor ϵ que utilizaremos como distancia máxima de separación entre un vértice del grafo y cada *Empty* de la escena (para considerar que están asociados).

Listado 17.1: Script de exportación de *NoEscapeDemo*

```
1  # Exportador NoEscape 1.1 (Adaptado a Blender 2.65)
2
3  import bpy, os, sys
4  import mathutils
5  from math import *
6  from bpy import *
7
8  FILENAME = "output.xml"       # Archivo XML de salida
9  GRAPHNAME = "Graph"           # Nombre del objeto Mesh del grafo
10 EPSILON = 0.01                # Valor de distancia Epsilon
11
12 # --- isclose ------------------------------------------------------
13 # Decide si un empty coincide con un vertice del grafo
14 # ------------------------------------------------------------------
15 def isclose(empty, coord):
16     xo, yo, zo = coord
17     xd, yd, zd = empty.location
18     v = mathutils.Vector((xo-xd, yo-yd, zo-zd))
19     if (v.length < EPSILON):
20         return True
21     else:
22         return False
23
24 # --- gettype -----------------------------------------------------
25 # Devuelve una cadena con el tipo del nodo del grafo
26 # -----------------------------------------------------------------
27 def gettype (dv, key):
28     obs = [ob for ob in bpy.data.objects if ob.type == 'EMPTY']
29     for empty in obs:
30         empName = empty.name
31         if ((empName.find("spawn") != -1) or
32             (empName.find("drain") != -1)):
33             if (isclose(empty, dv[key])):
34                 return 'type ="'+ empName[:-1] +'"'
35     return 'type=""'
```

C17

```
36
37 ID1 = ' '*2    # Identadores para el xml
38 ID2 = ' '*4    # Solo con proposito de obtener un xml "bonito"
39 ID3 = ' '*6
40 ID4 = ' '*8
41
42 graph = bpy.data.objects[GRAPHNAME]
43
44 dv = {}        # Diccionario de vertices
45 for vertex in graph.data.vertices:
46     dv[vertex.index+1] = vertex.co
47
48 de = {}        # Diccionario de aristas
49 for edge in graph.data.edges:           # Diccionario de aristas
50     de[edge.index+1] = (edge.vertices[0], edge.vertices[1])
51
52 file = open(FILENAME, "w")
53 std=sys.stdout
54 sys.stdout=file
55
56 print ("<?xml version='1.0' encoding='UTF-8'?>\n")
57 print ("<data>\n")
58
59 # --------- Exportacion del grafo ---------------------
60 print ("<graph>")
61 for key in dv.keys():
62     print (ID1 + '<vertex index="' + str(key) + '" '+
63            gettype(dv,key) +'>')
64     x,y,z = dv[key]
65     print (ID2 + '<x> %f</x> <y> %f</y> <z> %f</z>' % (x,y,z))
66     print (ID1 + '</vertex>')
67 for key in de.keys():
68     print (ID1 + '<edge>')
69     v1,v2 = de[key]
70     print (ID2 + '<vertex> %i</vertex> <vertex> %i</vertex>'
71            % (v1,v2))
72     print (ID1 + '</edge>')
73 print ("</graph>\n")
74
75 # --------- Exportacion de la camara -------------------
76 obs = [ob for ob in bpy.data.objects if ob.type == 'CAMERA']
77 for camera in obs:
78     camId = camera.name
79     camName = camId.split("_")[0]
80     camIndex = int(camId.split("_")[1])
81     camFrames = int (camId.split("_")[2])
82     print ('<camera index="%i" fps="%i">' %
83            (camIndex, bpy.data.scenes['Scene'].render.fps))
84     print (ID1 + '<path>')
85     for i in range (camFrames):
86         cFrame = bpy.data.scenes['Scene'].frame_current
87         bpy.data.scenes['Scene'].frame_set(cFrame+1)
88         x,y,z = camera.matrix_world.translation
89         qx,qy,qz,qw = camera.matrix_world.to_quaternion()
90         print (ID2 + '<frame index="%i">' % (i+1))
91         print (ID3 + '<position>')
92         print (ID4 + '<x> %f</x> <y> %f</y> <z> %f</z>' % (x,y,z))
93         print (ID3 + '</position>')
94         print (ID3 + '<rotation>')
95         print (ID4 + '<x> %f</x> <y> %f</y> <z> %f</z> <w> %f</w>' %
96                (qx,qy,qz,qw))
97         print (ID3 + '</rotation>')
98         print (ID2 + '</frame>')
99     print (ID1 + '</path>')
100    print ('</camera>')
101
102 print ("</data>")
103
104 file.close()
105 sys.stdout = std
```

El proceso de exportación del XML se realiza directamente *al vuelo*, sin emplear las facilidades (como el DOM o SAX de Python). La sencillez del XML descrito, así como la independencia de una instalación *completa* de Python hace que ésta sea la alternativa implementada en otros exportadores (como el exportador de Ogre, o el de Collada).

En las líneas `44,48` se crean dos diccionarios donde se guardarán las listas relativas a las coordenadas de los vértices del grafo y las aristas. Blender numera los vértices y aristas asociados a una malla poligonal comenzando en 0. En la descripción del XML indicamos por convenio que todos los índices comenzarían en 1, por lo que es necesario sumar 1 a los índices que nos devuelve Blender.

La malla asociada al objeto de nombre indicado en *GRAPHNAME* se obtiene mediante la llamada a *bpy.data.objects* (línea `42`). La línea `56` simplemente imprime la cabecera del XML.

En el bloque descrito por `59-73` se generan las entradas relativas a la definición del grafo, empleando los diccionarios de vértices *dv* y de aristas *de* creados anteriormente.

En la definición del XML (ver Capítulo 6 del Módulo 1), un nodo puede tener asociado un tipo. Para obtener el tipo de cada nodo utilizamos una función auxiliar llamada *gettype* (definida en las líneas `27-35`).

Muy Cerca!

Se emplea un valor Epsilon para calcular si un Empty ocupa la posición de un nodo del grafo porque las comparaciones de igualdad con valores de punto flotante pueden dar resultados incorrectos debido a la precisión.

En *gettype* se calcula la cadena correspondiente al tipo del nodo del grafo. Como hemos indicado anteriormente, el tipo del nodo se calcula según la distancia a los objetos Empty de la escena. Si el vértice está *muy cerca* de un Empty con subcadena "drain" o "spawn" en su nombre, le asignaremos ese tipo al vértice del grafo. En otro caso, el nodo del grafo será genérico. De este modo, en la línea `28` se obtiene en *obs* la lista de todos los objetos de tipo *Empty* de la escena. Para cada uno de estos objetos, si el nombre contiene alguna de las subcadenas indicadas y está suficientemente cerca (empleando la función auxiliar *isclose*), entonces devolvemos ese tipo (línea `34`) para el nodo pasado como argumento de la función (línea `27`). En otro caso, devolvemos la cadena vacía para tipo (que indica que no es un nodo especial del grafo) en la línea `35`.

La función auxiliar *isclose* (definida en las líneas `15-22` devolverá *True* si el empty está a una distancia menor que ϵ respecto del nodo del grafo. En otro caso, devolverá *False*. Para realizar esta comparación, simplemente creamos un vector restando las coordenadas del vértice con la posición del *Empty* y utilizamos directamente el atributo de longitud (línea `19`) de la clase *Vector* de la biblioteca *mathutils*.

La biblioteca *Mathutils* de Blender contiene una gran cantidad de funciones para trabajar con Matrices, Vectores, Cuaternios y un largo etcétera.

Trayectorias genéricas

Aunque en este ejemplo exportaremos trayectorias sencillas, el exportar la posición de la cámara en cada frame nos permite tener un control absoluto sobre la pose en cada instante. En la implementación en OGRE tendremos que utilizar algún mecanismo de interpolación entre cada pose clave.

La segunda parte del código (definido en las líneas `76-100`) se encarga de exportar las animaciones asociadas a las cámaras. Según el convenio explicado anteriormente, el propio nombre de la cámara codifica el índice de la cámara (ver línea `80`) y el número de frames que contiene su animación (ver línea `81`). En el caso de este ejemplo se han definido dos cámaras (ver Figura 17.4). La cámara del menú inicial es estática, por lo que define en la parte relativa a los frames un valor de 1 (no hay animación). La cámara del juego describirá rotaciones siguiendo una trayectoria circular.

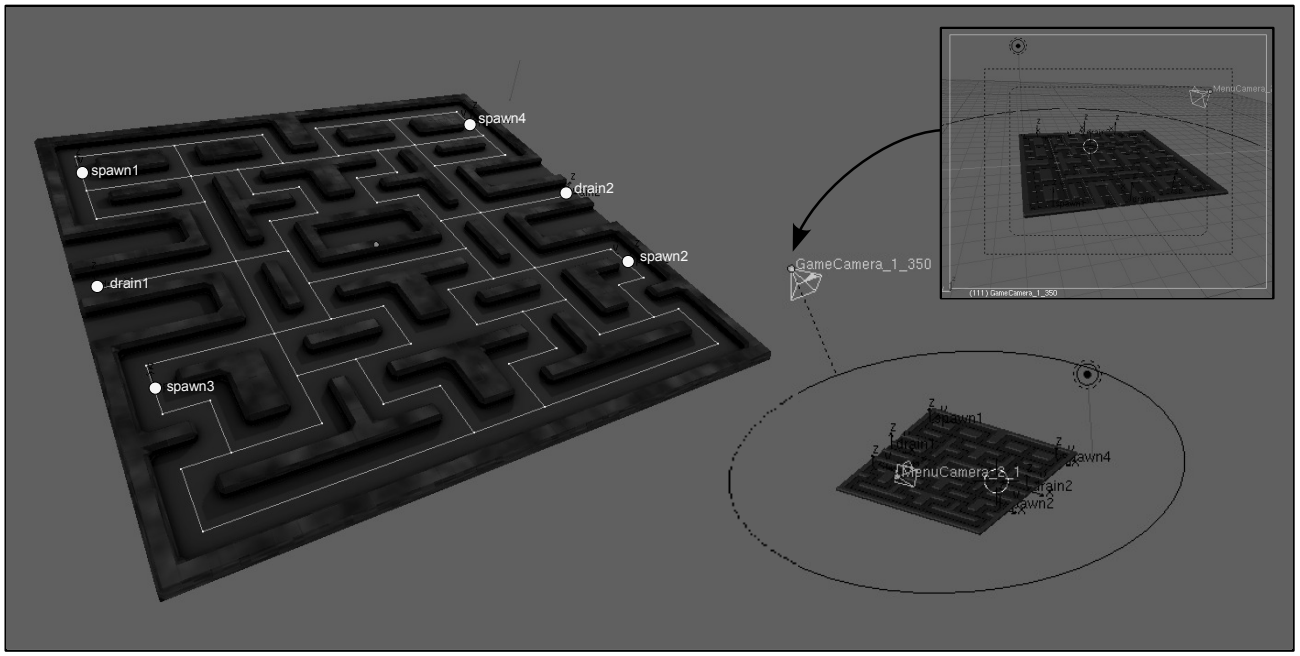


Figura 17.4: Definición del grafo asociado al juego *NoEscapeDemo* en Blender. En la imagen de la izquierda se han destacado la posición de los Emptys auxiliares para describir el tipo especial de algunos nodos (vértices) del grafo. La imagen de la derecha muestra las dos cámaras de la escena y la ruta descrita para la cámara principal del juego.

El número de frame se modifica directamente en el atributo *frame_current* del la escena actual (ver línea 86). Para cada frame se exporta la posición de la cámara (obtenida en la última columna de la matriz de transformación del objeto, en la línea 88), y el cuaternio de rotación (mediante una conversión de la matriz en 89). La Figura 17.5 muestra un fragmento del XML exportado relativo a la escena base del ejemplo. Esta exportación puede realizarse ejecutando el fichero .py desde línea de órdenes, llamando a Blender con el parámetro *-P <script.py>* (siendo script.py el nombre del script de exportación).

```
<?xml version='1.0' encoding='UTF-8'?>
<data>
<graph>
  <vertex index="1" type="">
    <x>-2.500000</x> <y>-2.499999</y> <z>0.000000</z>
  </vertex>
  ...
  <vertex index="57" type ="drain">
    <x>-4.687499</x> <y>0.312501</y> <z>0.000000</z>
  </vertex>
  ...
  <edge> <vertex>5</vertex> <vertex>15</vertex> </edge>
  <edge> <vertex>6</vertex> <vertex>28</vertex> </edge>
  ...
</graph>
<camera index="1" fps="25">
  <path>
    <frame index="1">
      <position>
       <x>1.077411</x> <y>-15.124874</y> <z>6.665801</z>
      </position>
      <rotation>
        <x>0.836699</x> <y>0.544365</y> <z>-0.004856</z>  <w>0.059819</w>
      </rotation>
    </frame>
    <frame index="2">
      ...
```

Figura 17.5: Fragmento del resultado de la exportación del XML del ejemplo.

Capítulo 18
Animación

Carlos González Morcillo

En este capítulo estudiaremos los fundamentos de la animación por computador, analizando los diversos métodos y técnicas de construcción, realizando ejemplos de composición básicos con Ogre. Se estudiará el concepto de *Animation State*, exportando animaciones definidas previamente en Blender.

18.1. Introducción

Animación?

El término *animación* proviene del griego *Anemos*, que es la base de la palabra latina *Animus* que significa *Dar aliento, dar vida*.

En su forma más simple, la animación por computador consiste en generar un conjunto de imágenes que, mostradas consecutivamente, producen sensación de movimiento. Debido al fenómeno de la *Persistencia de la Visión*, descubierto en 1824 por Peter Mark Roget, el ojo humano retiene las imágenes una vez vistas unos 40 ms. Siempre que mostremos imágenes a una frecuencia mayor, tendremos sensación de movimiento continuo[1]. Cada una de las imágenes que forman la secuencia animada recibe el nombre de *frame*[2].

La animación por computador cuenta con una serie de ventajas que no se dan en animación tradicional; por ejemplo, la animación puede producirse directamente desde modelos o conjuntos de ecuaciones que especifican el comportamiento dinámico de los objetos a animar.

[1] A frecuencias menores de 20hz se percibirá la discretización del movimiento, en un efecto de tipo estroboscópico.

[2] también, aunque menos extendido en el ámbito de la animación por computador se utiliza el término cuadro o fotograma

Cuando hablamos de técnicas de animación por computador nos referimos a sistemas de *control del movimiento*. Existen multitud de elementos que hacen el problema del control del movimiento complejo, y de igual forma, existen varias aproximaciones a la resolución de estas dificultades. La disparidad de aproximaciones y la falta de unidad en los convenios de nombrado, hacen que la categorización de las técnicas de animación por computador sea difícil. Una propuesta de clasificación de se realiza según el nivel de abstracción.

Así, se distingue entre sistemas de animación de *alto nivel*, que permiten al animador especificar el movimiento en términos generales (definen el comportamiento en términos de eventos y relaciones), mientras que los sistemas de *bajo nivel* requieren que el animador indique los parámetros de movimiento individualmente.

Si se desean animar objetos complejos (como por ejemplo la figura humana), es necesario, al menos, un control de jerarquía para reducir el número de parámetros que el animador debe especificar. Incluso en personajes digitales sencillos, como el mostrado en la figura 18.1 el esqueleto interno necesario tiene un grado de complejidad considerable. En concreto, este esqueleto tiene asociados 40 huesos con diversos modificadores aplicados individualmente.

Al igual que en los lenguajes de programación de alto nivel, las construcciones deben finalmente *compilarse* a instrucciones de bajo nivel. Esto implica que cualquier descripción paramétrica de alto nivel debe transformarse en descripciones de salida de bajo nivel. Este proceso proceso finalizará cuando dispongamos de todos los datos necesarios para todos los frames de la animación.

De esta forma, lo que se busca en el ámbito de la animación es ir subiendo de nivel, obteniendo sistemas de mayor nivel de abstracción. Las primeras investigaciones comenzaron estudiando las técnicas de interpolación de movimiento entre frames clave, utilizando Splines. Mediante este tipo de curvas, los objetos podían moverse de forma suave a lo largo de un camino en el espacio. Desde este punto, han aparecido multitud de técnicas de animación de medio nivel (como la animación basada en scripts, animación procedural, animación jerárquica basada en cinemática directa e inversa y los estudios de síntesis animación automática).

A continuación estudiaremos los principales niveles de abstracción relativos a la definición de animación por computador.

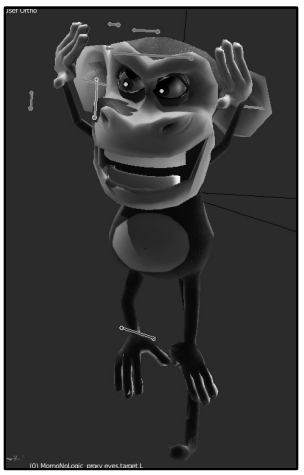


Figura 18.1: Uno de los personajes principales de YoFrankie, proyecto de videojuego libre de la Blender Foundation.

Técnicas básicas

Estas técnicas forman la base de los métodos de animación más avanzados, por lo que es imprescindible conocer su funcionamiento para emplear correctamente los métodos de animación basados en cinemática directa o inversa que estudiaremos más adelante.

18.1.1. Animación Básica

Lo fundamental en este nivel es cómo parametrizar los caminos básicos de movimiento en el espacio. Hay diversas alternativas (no excluyentes) como los sistemas de Script, sistemas de Frame Clave y animación dirigida por Splines.

- **Sistemas de Script**. Históricamente, los primeros sistemas de control del movimiento fueron los sistemas basados en scripts. Este tipo de sistemas requieren que el usuario escriba un guión en un lenguaje específico para animación y además, presuponen una habilidad por parte del animador de expresar la animación con el lenguaje de script. Este tipo de aproximación producen animaciones de baja calidad, dada la complejidad en la especificación de las acciones.
- **Sistema de Frame Clave**. Los sistemas de Frame Clave toman su nombre de la jerarquía de producción tradicional de Walt Disney. En estos sistemas, los animadores más experimentados diseñaban los fotogramas principales de cada secuencia. La producción se completaba con artistas jóvenes que añadían los frames intermedios[3].

[3] Realizando la técnica llamada ïn between"

Dependiendo del tipo de método que se utilice para el cálculo de los fotogramas intermedios estaremos ante una técnica de interpolación u otra. En general, suelen emplearse curvas de interpolación del tipo Spline, que dependiendo de sus propiedades de continuidad y el tipo de cálculo de los ajustes de tangente obtendremos diferentes resultados. En el ejemplo de la Figura 18.2, se han añadido claves en los frames 1 y 10 con respecto a la posición (localización) y rotación en el eje Z del objeto. El ordenador calcula automáticamente la posición y rotación de las posiciones intermedias. La figura muestra las posiciones intermedias calculadas para los frames 4 y 7 de la animación.

- **Animación Procedural**. En esta categoría, el control sobre el movimiento se realiza empleando procedimientos que definen explícitamente el movimiento como función del tiempo. La generación de la animación se realiza mediante descripciones físicas, como por ejemplo en visualizaciones científicas, simulaciones de fluidos, tejidos, etc... Estas técnicas de animación procedural serán igualmente estudiadas a continuación este módulo curso. La simulación dinámica, basada en descripciones matemáticas y físicas suele emplearse en animación de acciones secundarias. Es habitual contar con animaciones almacenadas a nivel de vértice simulando comportamiento físico en videojuegos. Sería el equivalente al *Baking* de animaciones complejas precalculadas.

Con base en estas técnicas fundamentales se definen los métodos de animación de alto nivel descritos a continuación.

18.1.2. Animación de Alto Nivel

Dentro de este grupo incluimos la animación mediante técnicas cinemáticas jerárquicas (como cinemática directa o inversa de figuras articuladas), las modernas aproximaciones de síntesis automática de animación o los sistemas de captura del movimiento. Estas técnicas específicas serán estudiadas en detalle en el Módulo 3 del curso, por lo que únicamente realizaremos aquí una pequeña descripción introductoria.

- **Cinemática Directa**. Empleando cinemática directa, el movimiento asociado a las articulaciones debe ser especificado explícitamente por el animador. En el ejemplo de la Figura 18.3, la animación del efector final vendría determinada indirectamente por la composición de transformaciones sobre la base, el hombro, el codo y la muñeca. Esto es, una estructura arbórea descendente. Esto es, dado el conjunto de rotaciones θ, obtenemos la posición del efector final X como $X = f(\theta)$.
- **Cinemática Inversa**. El animador define únicamente la posición del efector final. Mediante cinemática inversa se calcula la posición y orientación de todas las articulaciones de la jerarquía que consiguen esa posición particular del efector final mediante $\theta = f(X)$.

Una vez realizada esta introducción general a las técnicas de animación por computador, estudiaremos en la siguiente sección los tipos de animación y las características generales de cada uno de ellos en el motor gráfico Ogre.

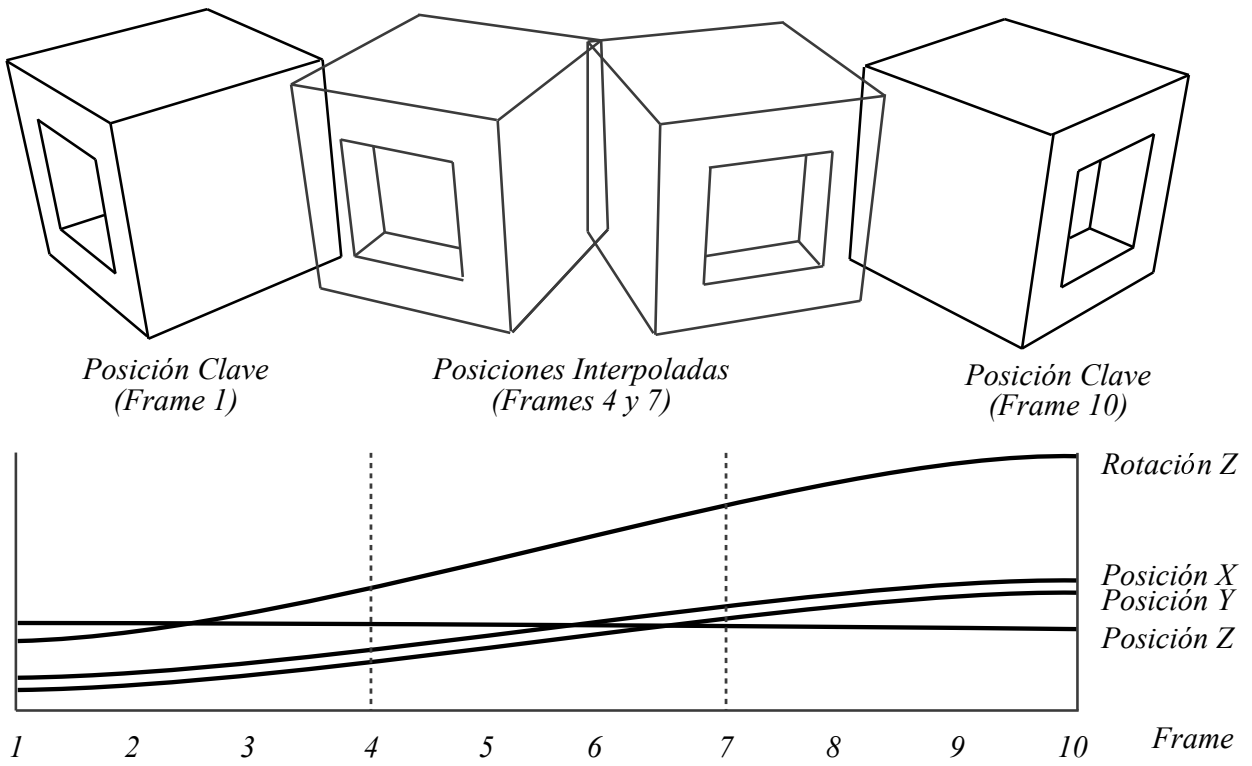


Figura 18.2: Ejemplo de uso de Frames Clave y Splines asociadas a la interpolación realizada.

18.2. Animación en Ogre

Ogre gestiona la posición de los elementos de la escena en cada frame. Esto significa que la escena es *redibujada* en cada frame. Ogre no mantiene ningún tipo de estructuras de datos con el estado anterior de los objetos de la escena. La animación se gestiona con dos aproximaciones; una basada en frames clave y otra mediante una variable (habitualmente se empleará el tiempo) utilizada como controlador. El modo más sencillo de crear animaciones es mediante el uso de una herramienta externa y reproducirlas posteriormente.

Como hemos señalado anteriormente, de modo general Ogre soporta dos modos de animación: basada en Frames Clave (*Keyframe Animation*) y basada en Controladores.

AnimationState

Cada animación de un *AnimationState* tiene asociado un nombre único mediante el que puede ser accedido.

18.2.1. Animación Keyframe

Ogre utiliza el término pista *Track* para referirse a un conjunto de datos almacenados en función del tiempo. Cada muestra realizada en un determinado instante de tiempo es un *Keyframe*. Dependiendo del tipo de *Keyframes* y el tipo de pista, se definen diferentes tipos de animaciones.

Las animaciones asociadas a una Entidad se representan en un *AnimationState*. Este objeto tiene una serie de propiedades que pueden ser consultadas:

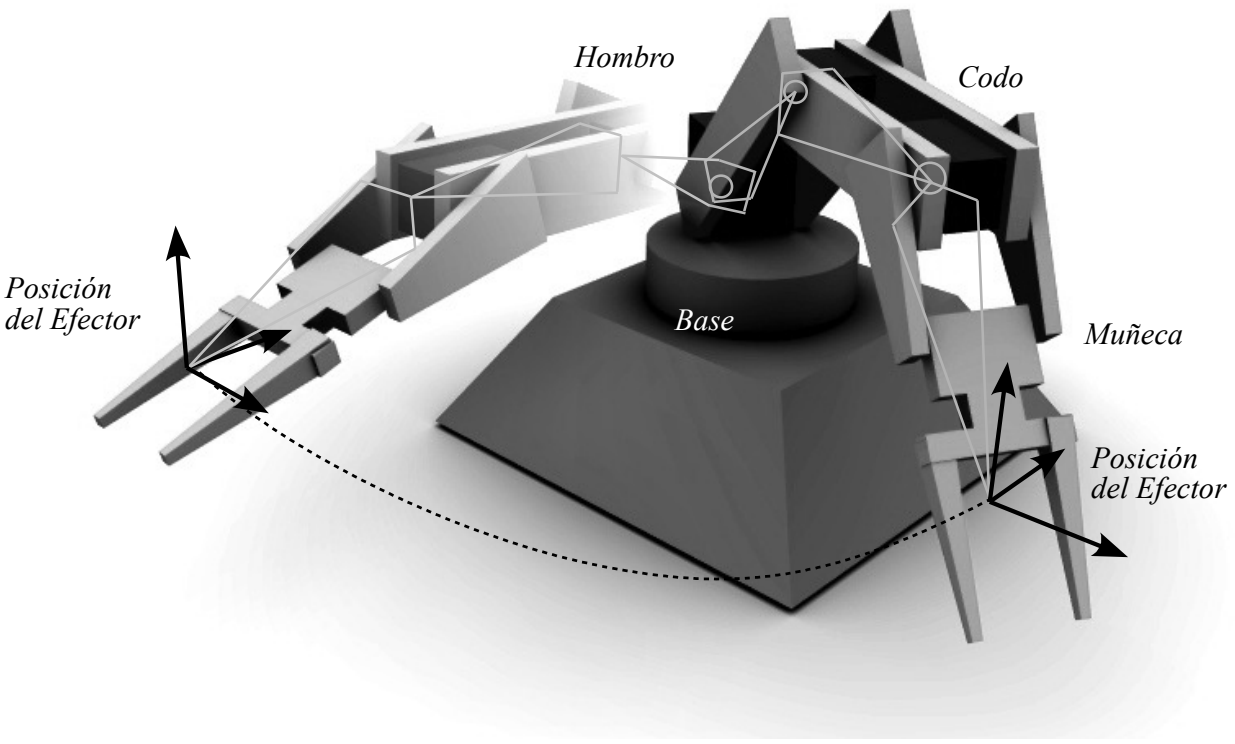


Figura 18.3: Ejemplo de uso de Frames Clave y Splines asociadas a la interpolación realizada.

- **Nombre**. Mediante la llamada a *getAnimationName se* puede obtener el nombre de la animación que está siendo utilizada por el *AnimationState*.
- **Activa**. Es posible activar y consultar el estado de una animación mediante las llamadas a *getEnabled* y *setEnabled*.
- **Longitud**. Obtiene en punto flotante el número de segundos de la animación, mediante la llamada a *getLength*.
- **Posición**. Es posible obtener y establecer el punto de reproducción actual de la animación mediante llamadas a *getTimePosition* y *setTimePosition*. El tiempo se establece en punto flotante en segundos.
- **Bucle**. La animación puede reproducirse en modo bucle (cuando llega al final continua reproduciendo el primer frame). Las llamadas a métodos son *getLoop* y *setLoop*.
- **Peso**. Este parámetro controla la mezcla de animaciones, que será descrita en la sección 18.4.

Tiempo

El tiempo transcurrido desde la última actualización puede especificarse con un valor o negativo (en el caso de querer retroceder en la animación). Este valor de tiempo simplemente se añade a la posición de reproducción de la animación.

El estado de la animación requiere que se le especifique el tiempo transcurrido desde la última vez que se actualizó.

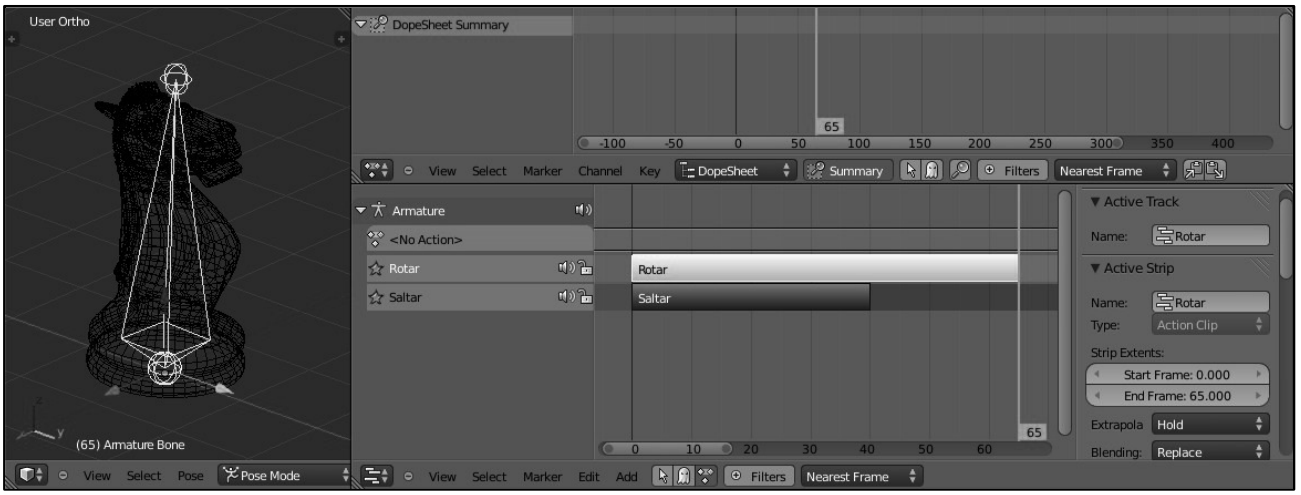


Figura 18.4: Principales ventanas de Blender empleadas en la animación de objetos. En el centro se encuentran las ventanas del *NLA Editor* y del *Dope Sheet*, que se utilizarán para definir acciones que serán exportadas empleando el script de Exportación a Ogre.

18.2.2. Controladores

La animación basada en frames clave permite animar mallas poligionales. Mediante el uso de controladores se posicionan los nodos, definiendo una función. El controlador permite modificar las propiedades de los nodos empleando un valor que se calcula en tiempo de ejecución.

18.3. Exportación desde Blender

En esta sección estudiaremos la exportación de animaciones de cuerpos rígidos. Aunque para almacenar los *AnimationState* es necesario asociar un esqueleto al objeto que se desea animar, no entraremos en detalles para la animación de jerarquías compuestas (personajes). Desde el punto de vista de la exportación en Ogre es similar.

Para exportar diferentes animaciones, será conveniente configurar el interfaz de Blender como se muestra en la Figura 18.4. A continuación describiremos brevemente las dos ventanas principales que utilizaremos en la exportación de animaciones:

El **NLA Editor** permite la edición de animación no lineal en Blender. Mediante esta técnica de composición, la animación final se crea mediante fragmentos de animación específicos (cada uno está especializado en un determinado movimiento) llamados *acciones*. Estas acciones pueden duplicarse, repetirse, acelerarse o decelerarse en el tiempo para formar la animación final.

El **Dope Sheet** (en versiones anteriores de Blender se denominaba *Action Editor*) permite el control de múltiples acciones simultáneamente, agrupándolas por tipo. Este editor está a un nivel de abstracción medio, entre las curvas IPO y el *NLA Editor* descrito anteriormente. Esta ventana permite trabajar muy rápidamente con los datos de la animación, desplazando la posición de los frames clave (y modificando así el *Timing* de cada acción). En el área de trabajo principal del *Dope Sheet* se muestran los canales de animación asociados a cada objeto. Cada *diamante* de las barras asociadas a cada canal se corresponden con un *keyframe* y pueden ser desplazadas individualmente o en grupos para modificar la temporización de cada animación.

Acciones

Las accciones permiten trabajar a un nivel de abstracción mayor que empleando curvas IPO (InterPOlation curve). Es conveniente emplear esta aproximación siempre que definamos animaciones complejas.

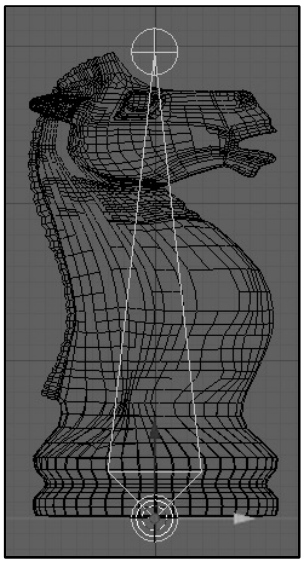


Figura 18.5: Ajuste del tamaño del hueso para que sea similar al objeto a animar.

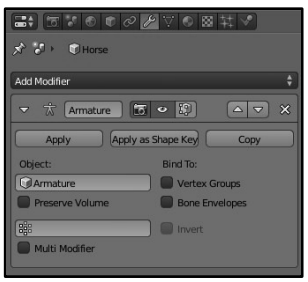


Figura 18.6: Opciones del Modificador *Armature*.

Figura 18.7: Ventanas de DopeSheet y NLA Editor.

Figura 18.8: Especificación del nombre de la acción en el Panel de Propiedades.

Las acciones aparecerán en el interfaz del exportador de Ogre, siempre que hayan sido correctamente creadas y estén asociadas a un hueso de un esqueleto. En la Figura 18.4 se han definido dos acciones llamadas "Saltar" y "Rotar" (la acción Rotar está seleccionada y aparece resaltada en la figura). A continuación estudiaremos el proceso de exportación de animaciones.

Como se ha comentado anteriormente, las animaciones deben exportarse empleando animación basada en esqueletos. En el caso más sencillo de animaciones de cuerpo rígido, bastará con crear un esqueleto de un único hueso y emparentarlo con el objeto que queremos animar.

Añadimos un esqueleto con un único hueso al objeto que queremos añadir mediante Shift A *Add/Armature/Single Bone*. Ajustamos el extremo superior del hueso para que tenga un tamaño similar al objeto a animar (como se muestra en la Figura 18.5). A continuación crearemos una relación de parentesco entre el objeto y el hueso del esqueleto, de forma que el objeto sea hijo de el esqueleto.

Este parentesco debe definirse en modo Pose . Con el hueso del esqueleto seleccionado, elegiremos el modo Pose en la cabecera de la ventana 3D (o bien empleando el atajo de teclado Control TAB). El hueso deberá representarse en color azul en el interfaz de Blender. Con el hueso en modo pose, ahora seleccionamos primero al caballo, y a continuación con Shift pulsado seleccionamos el hueso (se deberá elegir en color azul), y pulsamos Control P, eligiendo *Set Parent to ->Bone*. Si ahora desplazamos el hueso del esqueleto en modo pose, el objeto deberá seguirle.

Antes de crear las animaciones, debemos añadir un Modificador de objeto sobre la malla poligional. Con el objeto *Horse* seleccionado, en el panel *Object Modifiers* añadimos un modificador de tipo *Armature*. Este modificador se utiliza para animar las poses de personajes y objetos asociados a esqueletos. Especificando el esqueleto que utilizaremos con cada objeto podremos incluso deformar los vértices del propio modelo.

Como se muestra en la Figura 18.6, en el campo *Object* especificaremos el nombre del esqueleto que utilizaremos con el modificador ("*Armature*" en este caso). En el método de *Bind to* indicamos qué elementos se asociarán al esqueleto. Si queremos que los vértices del modelo se deformen en función de los huesos del esqueleto, activaremos **Vertex Groups** (en este ejemplo deberá estar desactivado). Mediante **Bone Envelopes** podemos indicar a Blender que utilice los límites de cada hueso para definir la deformación que se aplicará a cada hueso.

A continuación creamos un par de animaciones asociadas a este hueso. Recordemos que debemos trabajar en modo Pose , por lo que siempre el hueso debe estar seleccionado en color azul en el interfaz de Blender. Comenzaremos definiendo una acción que se llamará "*Saltar*". Añadiremos los frames clave de la animación (en este caso, hemos definido frames clave de *LocRot* en 1, 21 y 35).

Tras realizar esta operación, las ventanas de *DopeSheet* y *NLA Editor* mostrarán un aspecto como el de la Figura 18.7. La ventana *DopeSheet* permite modificar fácilmente las posiciones de los frames clave asociados al hueso. Pinchando y desplazando los rombos asociados a cada clave, podemos modificar el *timing* de la animación. En la ventana *NLA* podemos crear una acción (*Action Strip*), pinchando sobre el icono del copo de nieve . Tras esta operación, se habrá creado una acción definida mediante una barra amarilla.

Es posible indicar el nombre de la acción en el campo *Active Strip*, accesible pulsando la tecla N. Aparecerá un nuevo subpanel, como se muestra en la Figura 18.8, en la zona derecha de la ventana, donde se podrán configurar los parámetros asociados a la acción (nombre, frames, método de mezclado con otras acciones, etc...).

Siguiendo el mismo procedimiento definimos otra acción llamada "*Rotar*". Para que no tengan influencia las acciones previamente creadas, y no molesten a la hora de definir nuevos comportamientos, es posible *ocultarlas* pinchando sobre el botón situado a la derecha del nombre de la acción (ver Figura 18.8).

Es posible exportar diferentes animaciones asociadas a un objeto. El rango de definición de los frames puede ser diferente, como en el ejemplo estudiado en esta sección.

Mediante el atajo de teclado Alt A podemos reproducir la animación relativa a la acción que tenemos seleccionada. Resulta muy cómodo emplear una ventana de tipo *Timeline* para definir el intervalo sobre el que se crearán las animaciones.

A la hora de exportar las animaciones, procedemos como vimos en el Capítulo 11. En *File/ Export/ Ogre3D*, accederemos al exportador de Blender a Ogre3D. Es interesante activar las siguientes opciones de exportación:

- **Armature Animation**. Crea el archivo con extensión `.skeleton`, que contiene la definición jerárquica del conjunto de huesos, junto con su posición y orientación. Esta jerarquía incluye la influencia que tiene cada hueso sobre los vértices del modelo poligonal.
- **Independent Animations**. Exporta cada acción de forma independiente, de modo que no tienen influencia combinada.

El uso de las animaciones exportadas en Ogre es relativamente sencillo. En siguiente código muestra un ejemplo de aplicación de las animaciones previamente exportadas. Cuando se pulsa la tecla A o Z se reproduce la animación *Saltar* o *Rotar* hasta que finaliza. En la línea 18 se actualiza el tiempo transcurrido desde la última actualización. Cuando se pulsa alguna de las teclas anteriores, la animación seleccionada se reproduce desde el principio (línea 10).

Huesos y vértices

En este capítulo trabajaremos con huesos que tienen una influencia total sobre el objeto (es decir, influencia de 1.0 sobre todos los vértices del modelo).

En el ejemplo siguiente puede ocurrir que se interrumpa una animación por la selección de otra antes de finalizar el estado. Esto podía dar lugar a posiciones finales incorrectas. Por ejemplo, si durante la ejecución de la acción "*Saltar*" se pulsaba la tecla Z, el objeto se quedaba flotando en el aire ejecutando la segunda animación.

Listado 18.1: Fragmento de MyFrameListener.cpp.

```
1  if (_keyboard->isKeyDown(OIS::KC_A) ||
2      _keyboard->isKeyDown(OIS::KC_Z)) {
3    if (_keyboard->isKeyDown(OIS::KC_A))
4      _animState = _sceneManager->getEntity("Horse")->
5                   getAnimationState("Saltar");
6    else _animState = _sceneManager->getEntity("Horse")->
7                     getAnimationState("Rotar");
8    _animState->setEnabled(true);
9    _animState->setLoop(true);
10   _animState->setTimePosition(0.0);
11 }
12
13 if (_animState != NULL) {
14   if (_animState->hasEnded()) {
15     _animState->setTimePosition(0.0);
16     _animState->setEnabled(false);
17   }
18   else _animState->addTime(deltaT);
```

```
19 }
```

Es posible definir varias animaciones y gestionarlas mediante diferentes *AnimationState*. En el siguiente ejemplo se cargan dos animaciones fijas en el constructor del *FrameListener* (línea 3-8). Empleando las mismas teclas que en el ejemplo anterior, se resetean cada una de ellas, y se actualizan ambas de forma independiente (en las líneas 26 y 28). Ogre se encarga de mezclar sus resultados (incluso es posible aplicar transformaciones a nivel de nodo, pulsando la tecla R mientras se reproducen las animaciones del *AnimationState*).

Listado 18.2: Fragmento de MyFrameListener.cpp.

```
1  MyFrameListener::MyFrameListener() {
2    // ...
3     _animState = _sceneManager->getEntity("Horse")->
4                  getAnimationState("Saltar");
5    _animState->setEnabled(false);
6    _animState2 = _sceneManager->getEntity("Horse")->
7                  getAnimationState("Rotar");
8    _animState2->setEnabled(false);
9  }
10
11 bool MyFrameListener::frameStarted(const FrameEvent& evt) {
12   // ...
13   if (_keyboard->isKeyDown(OIS::KC_A)) {
14     _animState->setTimePosition(0.0);
15     _animState->setEnabled(true);
16     _animState->setLoop(false);
17   }
18
19   if (_keyboard->isKeyDown(OIS::KC_Z)) {
20     _animState2->setTimePosition(0.0);
21     _animState2->setEnabled(true);
22     _animState2->setLoop(false);
23   }
24
25   if (_animState->getEnabled() && !_animState->hasEnded())
26     _animState->addTime(deltaT);
27   if (_animState2->getEnabled() && !_animState2->hasEnded())
28     _animState2->addTime(deltaT);
29   // ...
```

Obviamente, será necesario contar con alguna clase de nivel superior que nos gestione las animaciones, y se encarge de gestionar los *AnimationState*, actualizándolos adecuadamente.

18.4. Mezclado de animaciones

En la sección anterior hemos utilizado dos canales de animación, con posibilidad de reproducción simultánea. En la mayoría de las ocasiones es necesario contar con mecanismos que permitan el mezclado controlado de animaciones (*Animation Blending*). Esta técnica fundamental se emplea desde hace años en desarrollo de videojuegos.

En la actualidad se emplean módulos específicos para el mezclado de animaciones. El uso de árboles de prioridad (*Priority Blend Tree*[4]) facilita al equipo artístico de una producción especificar con un alto nivel de detalle cómo se realizará la composición de las capas de animación.

El mezclado de animaciones requiere un considerable tiempo de CPU. La interpolación necesaria para mezclar los canales de animación en cada frame hace que el rendimiento del videojuego pueda verse afectado. Empleando interpoilación esférica SLERP, para cada elemento del esqueleto es necesario calcular varias operaciones costosas (cuatro *senos*, un *arccos*, y una raíz cuadrada).

A un alto nivel de abstracción, cada animación en Ogre tiene asociado un peso. Cuando se establece la posición dentro de la animación, puede igualmente modificarse el peso del canal de animación. Dependiendo del peso asignado a cada animación, Ogre se encargará de realizar la interpolación de todas las animaciones activas para obtener la posición final del objeto. A continuación definiremos una clase llamada *AnimationBlender* que se encargará de realizar la composición básica de capas de animación.

Las variables miembro privadas de la clase *AnimationBlender* contienen punteros a las animaciones de inicio y fin del mezclado (líneas 10 y 11), el tipo de transición deseado (que es un tipo enumerado definido en las líneas 3-6), y un booleano que indica si la animación se reproducirá en bucle.

Las variables públicas (definidas en 16-18) contienen el tiempo restante de reproducción de la pista actual, la duración (en segundos) del mezclado entre pistas y un valor booleano que indica si la reproducción ha finalizado.

La clase incorpora, además de los tres métodos principales que estudiaremos a continuación, una serie de métodos auxiliares que nos permiten obtener valores relativos a las variables privadas (líneas 24-27).

Figura 18.9: La clase de *AnimationBlender* (Mezclado de Animaciones) no tiene nada que ver con el prestigioso paquete de animación del mismo nombre.

Listado 18.3: AminationBlender.h

```
1  class AnimationBlender {
2   public:
3     enum BlendingTransition {
4       Switch,  // Parar fuente y reproduce destino
5       Blend    // Cross fade (Mezclado suave)
6     };
7
8   private:
9     Entity *mEntity;                 // Entidad a animar
10    AnimationState *mSource;         // Animacion inicio
11    AnimationState *mTarget;         // Animacion destino
12    BlendingTransition mTransition;  // Tipo de transicion
13    bool mLoop;                      // Animacion en bucle?
14
15  public:
16    Real mTimeleft;    // Tiempo restante de la animacion (segundos)
17    Real mDuration;    // Tiempo invertido en el mezclado (segundos)
18    bool mComplete;    // Ha finalizado la animacion?
19
20    AnimationBlender( Entity *);
21    void blend(const String &anim, BlendingTransition transition,
22            Real duration, bool l=true);
23    void addTime(Real);
24    Real getProgress() { return mTimeleft/mDuration; }
25    AnimationState *getSource() { return mSource; }
26    AnimationState *getTarget() { return mTarget; }
27    bool getLoop() { return mLoop; }
28  };
```

[4]Más información sobre cómo se implementó el motor de mezclado de animaciones del *MechWarrior* en `http://www.gamasutra.com/view/feature/3456/`

Veamos a continuación los principales métodos de la clase, declarados en las líneas 20-23 del listado anterior.

Listado 18.4: AminationBlender.cpp (Constructor)

```
1  AnimationBlender::AnimationBlender(Entity *ent) : mEntity(ent) {
2    AnimationStateSet *set = mEntity->getAllAnimationStates();
3    AnimationStateIterator it = set->getAnimationStateIterator();
4    // Inicializamos los AnimationState de la entidad
5    while(it.hasMoreElements()) {
6      AnimationState *anim = it.getNext();
7      anim->setEnabled(false);
8      anim->setWeight(0);
9      anim->setTimePosition(0);
10   }
11   mSource = NULL;  mTarget = NULL;  mTimeleft = 0;
12 }
```

En el constructor de la clase inicializamos todas las animaciones asociadas a la entidad (recibida como único parámetro). Mediante un iterador (línea 6) desactivamos todos los *AnimationState* asociados (línea 7), y reseteamos la posición de reproducción y su peso asociado para su posterior composición (líneas 8-9).

La clase *AnimationBlender* dispone de un método principal para *cargar* animaciones, indicando (como segundo parámetro) el tipo de mezcla que quiere realiarse con la animación que esté reproduciéndose. La implementación actual de la clase admite dos modos de mezclado; un efecto de mezclado suave tipo *cross fade* y una transición básica de intercambio de canales.

La transición de tipo *Blend* implementa una transición simple lineal de tipo *Cross Fade*. En la Figura 18.10 representa este tipo de transición, donde el primer canal de animación se mezclará de forma suave con el segundo, empleando una combinación lineal de pesos.

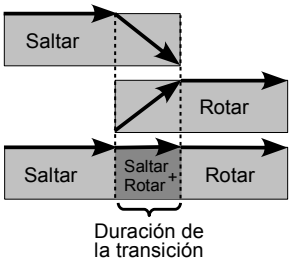


Figura 18.10: Efecto de transición tipo *Cross Fade*. En el tiempo especificado en la duración de la transición ambos canales de animación tienen influencia en el resultado final.

Listado 18.5: AminationBlender.cpp (Blend)

```
1  void AnimationBlender::blend (const String &anim,
2       BlendingTransition transition, Real duration, bool l) {
3
4    AnimationState *newTarget = mEntity->getAnimationState(anim);
5    newTarget->setLoop(l);
6    mTransition = transition;
7    mDuration = duration;
8    mLoop = l;
9
10   if ((mTimeleft<=0) || (transition == AnimationBlender::Switch)){
11     // No hay transicion (finalizo la anterior o Switch)
12     if (mSource != NULL) mSource->setEnabled(false);
13     mSource = newTarget;                    // Establecemos la nueva
14     mSource->setEnabled(true);
15     mSource->setWeight(1);                  // Con maxima influencia
16     mSource->setTimePosition(0);            // Reseteamos la posicion
17     mTimeleft = mSource->getLength();       // Duracion del AnimState
18     mTarget = NULL;
19   }
20   else {                                    // Hay transicion suave
21     if (mSource != newTarget) {
22       mTarget = newTarget;                  // Nuevo destino
23       mTarget->setEnabled(true);
24       mTarget->setWeight(0);                // Cambia peso en addTime
25       mTarget->setTimePosition(0);
26     }
```

```
27    }
28  }
```

Aunque la implementación actual utiliza una simple combinación lineal de pesos, es posible definir cualquier función de mezcla. Bastará con modificar la implementación del método *addTime* que veremos a continuación.

Por su parte, el método de mezclado de tipo *Switch* realiza un intercambio directo de los dos canales de animación (ver Figura 18.11). Este método es el utilizado igualmente si el canal que se está reproduciendo actualmente ha finalizado (ver línea 10 del listado anterior).

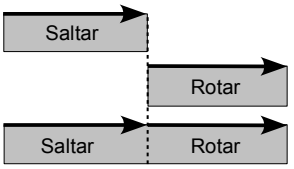


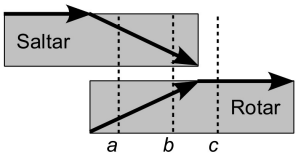
Figura 18.11: Transición de tipo *Intercambio*. El parámetro de duración, aunque se especifique, no tiene ningún efecto.

La implementación del método anterior tiene en cuenta el canal de animación que se está reproduciendo actualmente. Si el canal a mezclar es igual que el actual, se descarta (línea 21). En otro caso, se añade como objetivo, activando el nuevo canal pero estableciendo su peso a 0 (línea 24). La mezcla efectiva de las animaciones (el cambio de peso asociado a cada canal de animación) se realiza en el método *addTime*, cuyo listado se muestra a continuación.

Listado 18.6: AminationBlender.cpp (addTime)

```
1  void AnimationBlender::addTime(Real time) {
2    if (mSource == NULL) return;   // No hay fuente
3    mSource->addTime(time);
4    mTimeleft -= time;
5    mComplete = false;
6    if ((mTimeleft <= 0) && (mTarget == NULL)) mComplete = true;
7
8    if (mTarget != NULL) {  // Si hay destino
9      if (mTimeleft <= 0) {
10       mSource->setEnabled(false);
11       mSource->setWeight(0);
12       mSource = mTarget;
13       mSource->setEnabled(true);
14       mSource->setWeight(1);
15       mTimeleft = mSource->getLength();
16       mTarget = NULL;
17     }
18     else {   // Queda tiempo en Source... cambiar pesos
19       Real weight = mTimeleft / mDuration;
20       if (weight > 1) weight = 1.0;
21       mSource->setWeight(weight);
22       mTarget->setWeight(1.0 - weight);
23       if (mTransition == AnimationBlender::Blend)
24         mTarget->addTime(time);
25     }
26   }
27   if ((mTimeleft <= 0) && mLoop) mTimeleft = mSource->getLength();
28 }
```

Al igual que en el uso de animaciones básicas en Ogre, el método *addTime* debe ser llamado cada vez que se redibuje la escena, indicando el tiempo transcurrido desde la última actualización. La clase *AnimationBlender* se encargará a su vez de ejecutar el método *addTime* de los canales de animación activos (líneas 3 y 24). El método *addTime* lleva internamente la cuenta del tiempo que le queda al canal de reproducción al canal de animación. Cuando el tiempo es menor o igual que el tiempo empleado para la transición, se calculará el peso que se asignará a cada canal (líneas 19-22). El peso se calcula empleando una sencilla combinación lineal, de modo que el peso total para ambos canales en cualquier instante debe ser igual a uno (ver Figura 18.12).



a) W(*Saltar*)=0.7 W(*Rotar*)=0.3
b) W(*Saltar*)=0.2 W(*Rotar*)=0.8
c) W(*Saltar*)=0.0 W(*Rotar*)=1.0

Figura 18.12: Cálculo del peso de cada animación basándose en la duración de la transición. En los instantes de tiempo *a*, *b* y *c* se muestra el valor del peso asociado a cada canal de animación.

No es necesario que el peso final combinado de todas las animaciones sea igual a uno. No obstante, es conveniente que la suma de todos los canales de animación estén normalizados y sean igual a 1.

Cuántas animaciones?

La cantidad de animaciones necesarias para aplicar correctamente las técnicas de *Animation Blending* pueden ser muy elevadas. Por ejemplo, en *MechWarrior* de Microsoft cada robot tenía asociadas más de 150 animaciones.

En el caso de asignar pesos negativos, la animación se reproducirá empleando curvas de animación invertidas.

La composición de animaciones suele ser una tarea compleja. En el ejemplo desarrollado para ilustrar el uso de la clase *AnimationBlender* se han utilizado únicamente dos animaciones, y se han utilizado tiempos de transición fijos en todos los casos. En producción es importante trabajar las transiciones y las curvas de animación para obtener resultados atractivos.

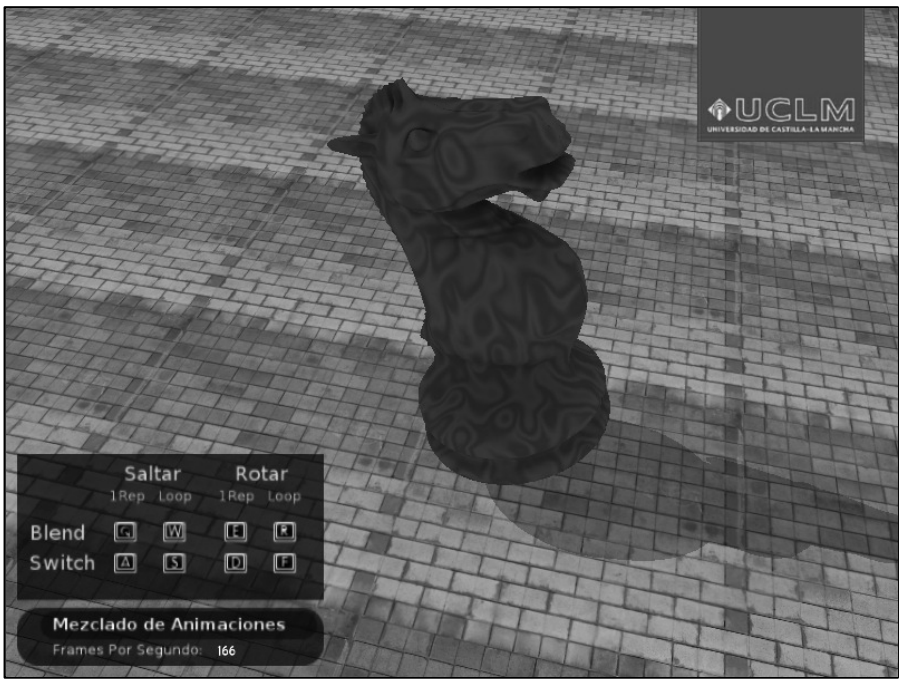


Figura 18.13: Ejemplo de aplicación que utiliza la clase *AnimationBlender*. Mediante las teclas indicadas en el interfaz es posible componer las animaciones exportadas empleandos los dos modos de transición implementados.

El mezclado de animaciones que son muy diferentes provoca resultados extraños. Es mejor combinar canales de animación que son similares, de forma que la mezcla funciona adecuadamente. En el ejemplo de esta sección se mezclan animaciones muy diferentes, para mostrar un caso extremo de uso de la clase. En entornos de producción, suelen definirse puntos de conexión entre animaciones, de modo que la clase de *Blending* se espera hasta alcanzar uno de esos puntos para realizar la mezcla. De este modo, se generan un alto número de animaciones para garantizar que las mezclas funcionarán correctamente sin comportamientos extraños.

Es importante realizar pruebas de la correcta composición y mezclado de animaciones en las primeras etapas del desarrollo del juego. De otra forma, podemos sufrir desagradables sorpresas cuando se acerca la deadline del proyecto. Puede resultar complicado ajustar en código la mezcla de animaciones, por lo que suelen desarrollarse scripts de exportación adicionales para indicar los puntos adecuados en cada animación donde puede componerse con el resto.

El siguiente listado muestra un ejemplo de uso de la clase. En el constructor del *FrameListener* se crea una variable miembro de la clase que contendrá un puntero al objeto *AnimBlender*. Cada vez que se actualiza el frame, llamamos al método *addTime* (ver línea 11) que actualizará los canales de animación convenientes. En este ejemplo se han utilizado las 8 teclas descritas en la Figura 18.13 para mezclar las dos animaciones exportadas en Blender al inicio del capítulo.

Listado 18.7: Uso de AnimationBlender

```
1  if (_keyboard->isKeyDown(OIS::KC_Q))
2    _animBlender->blend("Saltar",AnimationBlender::Blend,0.5, false);
3  if (_keyboard->isKeyDown(OIS::KC_R))
4    _animBlender->blend("Rotar", AnimationBlender::Blend, 0.5, true);
5
6  if (_keyboard->isKeyDown(OIS::KC_S))
7    _animBlender->blend("Saltar", AnimationBlender::Switch, 0, true);
8  if (_keyboard->isKeyDown(OIS::KC_D))
9    _animBlender->blend("Rotar", AnimationBlender::Switch, 0, false);
10
11 _animBlender->addTime(deltaT);
```

En este capítulo hemos estudiado los usos fundamentales de la animación de cuerpo rígido. En el módulo 3 del curso estudiaremos algunos aspectos avanzados, como la animación de personajes empleando esqueletos y aproximaciones de cinemática inversa y el uso de motores de simulación física para obtener, de forma automática, animaciones realistas.

Capítulo 19

Simulación Física

Carlos González Morcillo

En prácticamente cualquier videojuego (tanto 2D como 3D) es necesaria la detección de colisiones y, en muchos casos, la simulación realista de dinámica de cuerpo rígido. En este capítulo estudiaremos la relación existente entre sistemas de detección de colisiones y sistemas de simulación física, y veremos algunos ejemplos de uso del motor de simulación física libre *Bullet*.

19.1. Introducción

Figura 19.1: "*Anarkanoid, el machacaladrillos sin reglas* es un juego tipo *Breakout* donde la simulación física se reduce a una detección de colisiones 2D.

Cuerpo Rígido

Definimos un *cuerpo rígido* como un objeto sólido ideal, infinitamente duro y no deformable.

La mayoría de los videojuegos requieren en mayor o menor medida el uso de técnicas de detección de colisiones y simulación física. Desde un videojuego clásico como *Arkanoid*, hasta modernos juegos automovilísticos como *Gran Turismo* requieren definir la interacción de los elementos en el mundo físico.

El motor de *simulación física* puede abarcar una amplia gama de características y funcionalidades, aunque la mayor parte de las veces el término se refiere a un tipo concreto de simulación de la **dinámica de cuerpos rígidos**. Esta dinámica se encarga de determinar el movimiento de estos cuerpos rígidos y su interacción ante la influencia de fuerzas.

En el mundo real, los objetos no pueden pasar a través de otros objetos (salvo casos específicos convenientemente documentados en la revista *Más Allá*). En nuestro videojuego, a menos que tengamos en cuenta las colisiones de los cuerpos, tendremos el mismo efecto. El *sistema de detección de colisiones*, que habitualmente es un módulo del motor de simulación física, se encarga de calcular estas relaciones, determinando la relación espacial existente entre cuerpos rígidos.

La mayor parte de los videojuegos actuales incorporan ciertos elementos de simulación física básicos. Algunos títulos se animan a incorporar ciertos elementos complejos como simulación de telas, cuerdas, pelo o fluidos. Algunos elementos de simulación física son precalculados y almacenados como animaciones estáticas, mientras que otros necesitan ser calculados en tiempo real para conseguir una integración adecuada.

Como hemos indicado anteriormente, las tres tareas principales que deben estar soportadas por un motor de simulación física son la detección de colisiones, su resolución (junto con otras restricciones de los objetos) y calcular la actualización del mundo tras esas interacciones. De forma general, las características que suelen estar presentes en motores de simulación física son:

- Detección de colisiones entre objetos dinámicos de la escena. Esta detección podrá ser utilizada posteriormente por el módulo de simulación dinámica.
- Cálculo de lineas de visión y tiro parabólico, para la simulación del lanzamiento de proyectiles en el juego.
- Definición de geometría estática de la escena (cuerpos de colisión) que formen el escenario del videojuego. Este tipo de geometría puede ser más compleja que la geometría de cuerpos dinámicos.
- Especificación de fuerzas (tales como viento, rozamiento, gravedad, etc...), que añadirán realismo al videjuego.
- Simulación de destrucción de objetos: paredes y objetos del escenario.
- Definición de diversos tipos de articulaciones, tanto en elementos del escenario (bisagras en puertas, raíles...) como en la descripción de las articulaciones de personajes.
- Especificación de diversos tipos de motores y elementos generadores de fuerzas, así como simulación de elementos de suspensión y muelles.
- Simulación de fluidos, telas y cuerpos blandos (ver Figura 19.2).

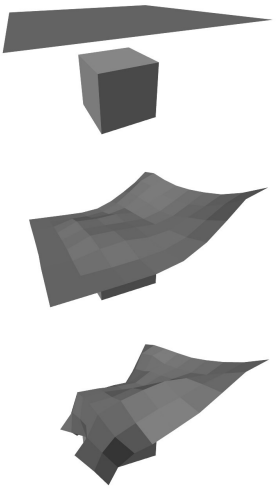


Figura 19.2: Tres instantes en la simulación física de una tela sobre un cubo. Simulación realizada con el motor Bullet.

19.1.1. Algunos Motores de Simulación

El desarrollo de un motor de simulación física desde cero es una tarea compleja y que requiere gran cantidad de tiempo. Afortunadamente existen gran variedad de motores de simulación física muy robustos, tanto basados en licencias libres como comerciales. A continuación se describirán brevemente algunas de las bibliotecas más utilizadas:

- **Bullet**. Bullet es una biblioteca de simulación física ampliamente utilizada tanto en la industria del videojuego como en la síntesis de imagen realista (Blender, Houdini, Cinema 4D y LightWave las utilizan internamente). Bullet es multiplataforma, y se distribuye bajo una licencia libre zlib compatible con GPL. Estudiaremos con más detalle este motor, junto con su uso en Ogre, en la Sección 19.5.
- **ODE**. ODE (*Open Dynamics Engine*) `www.ode.org` es un motor de simulación física desarrollado en C++ bajo doble licencias BSD y LGPL. El desarrollo de ODE comenzó en el 2001, y ha sido utilizado como motor de simulación física en multitud de éxitos mundiales, como el aclamado videojuego multiplataforma *World of Goo*, *BloodRayne 2* (PlayStation 2 y Xbox), y *TitanQuest* (Windows). Ogre cuenta igualmente con un wrapper para utilizar este motor de simulación física.

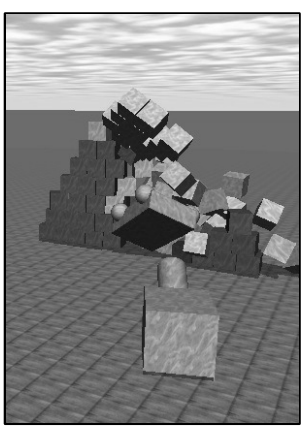


Figura 19.3: Ejemplo de simulación física con ODE (demo de la distribución oficial), que incorpora el uso de motores y diferentes geometrías de colisión.

Figura 19.4: Logotipo del motor de simulación físico estrella en el mundo del software privativo.

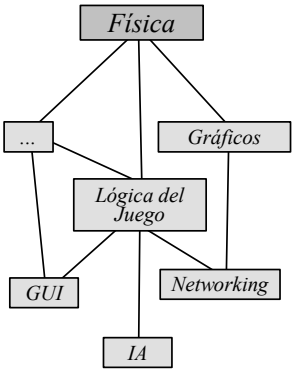


Figura 19.5: El motor de simulación física está directamente relacionado con otros módulos del videojuego. Esta dependencia conlleva una serie de dificultades que deben tenerse en cuenta.

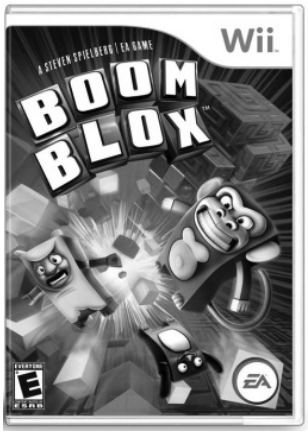


Figura 19.7: La primera incursión de Steven Spielberg en el mundo de los videojuegos fue en 2008 con *Boom Blox*, un título de Wii desarrollado por Electronic Arts con una componente de simulación física crucial para la experiencia del jugador.

- **PhysX**. Este motor privativo, es actualmente mantenido por NVidia con aceleración basada en hardware (mediante unidades específicas de procesamiento físico PPUs *Physics Processing Units* o mediante núcleos CUDA. Las tarjetas gráficas con soporte de CUDA (siempre que tengan al menos 32 núcleos CUDA) pueden realizar la simulación física en GPU. Este motor puede ejecutarse en multitud de plataformas como PC (GNU/Linux, Windows y Mac), PlayStation 3, Xbox y Wii. El SDK es gratuito, tanto para proyectos comerciales como no comerciales. Existen multitud de videojuegos comerciales que utilizan este motor de simulación. Gracias al wrapper *NxOgre* se puede utilizar este motor en Ogre.
- **Havok**. El motor Havok se ha convertido en el estándar de facto en el mundo del software privativo, con una amplia gama de características soportadas y plataformas de publicación (PC, Videoconsolas y Smartphones). Desde que en 2007 Intel comprara la compañía que originalmente lo desarrolló, Havok ha sido el sistema elegido por más de 150 videojuegos comerciales de primera línea. Títulos como *Age of Empires*, *Killzone 2 & 3*, *Portal 2* o *Uncharted 3* avalan la calidad del motor.

Existen algunas bibliotecas específicas para el cálculo de colisiones (la mayoría distribuidas bajo licencias libres). Por ejemplo, *I-Collide*, desarrollada en la Universidad de Carolina del Norte permite calcular intersecciones entre volúmenes convexos. Existen versiones menos eficientes para el tratamiento de formas no convexas, llamadas *V-Collide* y *RAPID*. Estas bibliotecas pueden utilizarse como base para la construcción de nuestro propio conjunto de funciones de colisión para videojuegos que no requieran funcionalidades físicas complejas.

19.1.2. Aspectos destacables

El uso de un motor de simulación física en el desarrollo de un videojuego conlleva una serie de aspectos que deben tenerse en cuenta relativos al diseño del juego, tanto a nivel de jugabilidad como de módulos arquitectónicos:

- **Predictibilidad**. El uso de un motor de simulación física afecta a la predictibilidad del comportamiento de sus elementos. Además, el ajuste de los parámetros relativos a la definición de las características físicas de los objetos (coeficientes, constantes, etc...) son difíciles de visualizar.
- **Realización de pruebas**. La propia naturaleza caótica de las simulaciones (en muchos casos no determinista) dificulta la realización de pruebas en el videojuego.
- **Integración**. La integración con otros módulos del juego puede ser compleja. Por ejemplo, ¿qué impacto tendrá en la búsqueda de caminos el uso de simulaciones físicas? ¿cómo garantizar el determinismo en un videojuego multijugador?.
- **Realismo gráfico**. El uso de un motor de simulación puede dificultar el uso de ciertas técnicas de representación realista (como por ejemplo el precálculo de la iluminación con objetos que pueden ser destruidos). Además, el uso de cajas límite puede producir ciertos resultados poco realistas en el cálculo de colisiones.

Figura 19.6: Gracias al uso de PhysX, las baldosas del suelo en *Batman Arkham Asylum* pueden ser destruidas (derecha). En la imagen de la izquierda, sin usar PhysX el suelo permanece inalterado, restando realismo y espectacularidad a la dinámica del juego.

- **Exportación**. La definición de objetos con propiedades físicas añade nuevas variables y constantes que deben ser tratadas por las herramientas de exportación de los datos del juego. En muchas ocasiones es necesario además la exportación de diferentes versiones de un mismo objeto (una versión de alta poligonalización, la versión de colisión, una versión destructible, etc).
- **Interfaz de Usuario**. Es necesario diseñar interfaces de usuario adaptados a las capacidades físicas del motor (¿cómo se especifica la fuerza y la dirección de lanzamiento de una granada?, ¿de qué forma se interactúa con objetos que pueden recogerse del suelo?).

19.1.3. Conceptos Básicos

A principios del siglo XVII, Isaac Netwon publicó las tres **leyes fundamentales del movimiento**. A partir de estos tres principios se explican la mayor parte de los problemas de dinámica relativos al movimiento de cuerpos y forman la base de la mecánica clásica. Las tres leyes pueden resumirse como:

1. Un cuerpo tiende a mantenerse en reposo o a continuar moviéndose en línea recta a una velocidad constante a menos que actúe sobre él una fuerza externa. Esta ley resume el concepto de inercia.
2. El cambio de movimiento es proporcional a la fuerza motriz aplicada y ocurre según la línea recta a lo largo de la que se aplica dicha fuerza.
3. Para cada fuerza que actúa sobre un cuerpo ocurre una reacción igual y contraria. De este modo, las acciones mutuas de dos cuerpos siempre son iguales y dirigidas en sentido opuesto.

En el estudio de la dinámica resulta especialmente interesante la segunda ley de Newton, que puede ser escrita como

$$F = m \times a \tag{19.1}$$

donde F es la fuerza resultante que actúa sobre un cuerpo de masa m, y con una aceleración lineal a aplicada sobre el centro de gravedad del cuerpo.

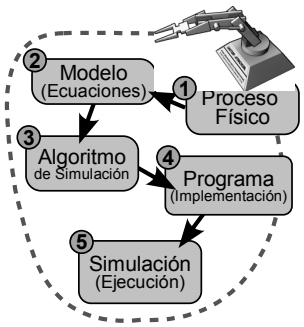


Figura 19.8: Etapas en la construcción de un motor de simulación física.

Desde el punto de vista de la mecánica en videojuegos, la **masa** puede verse como una medida de la resistencia de un cuerpo al movimiento (o al cambio en su movimiento). A mayor masa, mayor resistencia para el cambio en el movimiento. Según la segunda ley de Newton que hemos visto anteriormente, podemos expresar que $a = F/m$, lo que nos da una impresión de cómo la masa aplica resistencia al movimiento. Así, si aplicamos una **fuerza** constante e incrementamos la masa, la aceleración resultante será cada vez menor.

El **centro de masas** (o de **gravedad**) de un cuerpo es el punto espacial donde, si se aplica una fuerza, el cuerpo se desplazaría sin aplicar ninguna rotación.

Un **sistema dinámico** puede ser definido como cualquier colección de elementos que cambian sus propiedades a lo largo del tiempo. En el caso particular de las simulaciones de cuerpo rígido nos centraremos en el cambio de posición y rotación.

Así, nuestra **simulación** consistirá en la ejecución de un modelo matemático que describe un sistema dinámico en un ordenador. Al utilizar modelos, se simplifica el sistema real, por lo que la simulación no describe con total exactitud el sistema simulado.

Habitualmente se emplean los términos de interactividad y tiempo real de modo equivalente, aunque no lo son. Una **simulación interactiva** es aquella que consigue una tasa de actualización suficiente para el control por medio de una persona. Por su parte, una **simulación en tiepo real** garantiza la actualización del sistema a un número fijo de frames por segundo. Habitualmente los motores de simulación física proporcionan tasas de frames para la simulación interactiva, pero no son capaces de garantizar Tiempo Real.

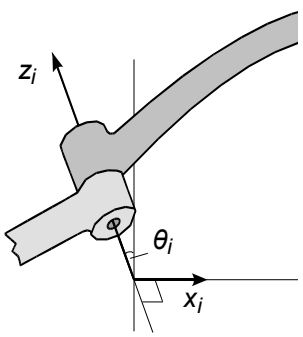


Figura 19.9: Un controlador puede intentar que se mantenga constante el ángulo θ_i formado entre dos eslabones de un brazo robótico.

En multitud de ocasiones es necesario definir restricciones que definan límites a ciertos aspectos de la simulación. Los **controladores** son elementos que generan entradas a la simulación y que tratan de controlar el comportamiento de los objetos que están siendo simulados. Por ejemplo, un controlador puede intentar mantener constante el ángulo entre dos eslabones de una cadena robótica (ver Figura 19.9).

Test de Intersección

En realidad, el *Sistema de Detección de Colisiones* puede entenderse como un módulo para realizar pruebas de intersección complejas.

19.2. Sistema de Detección de Colisiones

La responsabilidad principal del *Sistema de Detección de Colisiones* (SDC) es calcular cuándo *colisionan* los objetos de la escena. Para calcular esta *colisión*, los objetos se representan internamente por una forma geométrica sencilla (como esferas, cajas, cilindros...).

Además de comprobar si hubo colisión entre los objetos, el SDC se encarga de proporcionar información relevante al resto de módulos del simulador físico sobre las propiedades de la colisión. Esta información se utiliza para evitar efectos indeseables, como la penetración de un objeto en otro, y consegir la estabilidad en la simulación cuando el objeto llega a la posición de equilibrio.

19.2.1. Formas de Colisión

Como hemos comentado anteriormente, para calcular la colisión entre objetos, es necesario proporcionar una representación geométrica del cuerpo que se utilizará para calcular la colisión. Esta representación interna se calculará para determinar la posición y orientación del objeto en el mundo. Estos datos, con una descripción matemática mucho más simple y eficiente, son diferentes de los que se emplean en la representación visual del objeto (que cuentan con un mayor nivel de detalle).

Habitualmente se trata de simplificar al máximo la forma de colisión. Aunque el SDC soporte objetos complejos, será preferible emplear tipos de datos simples, siempre que el resultado sea aceptable. La Figura 19.10 muestra algunos ejemplos de aproximación de formas de colisión para ciertos objetos del juego.

Multitud de motores de simulación física separan la forma de colisión de la transformación interna que se aplica al objeto. De esta forma, como muchos de los objetos que intervienen en el juego son dinámicos, basta con aplicar la transformación a la forma de un modo computacionalmente muy poco costoso. Además, separando la transformación de la forma de colisión es posible que varias entidades del juego compartan la misma forma de colisión.

Algunos motores de simulación física permiten compartir la misma descripción de la forma de colisión entre entidades. Esto resulta especialmente útil en juegos donde la forma de colisión es compleja, como en simuladores de carreras de coches.

Como se muestra en la Figura 19.10, las entidades del juego pueden tener diferentes formas de colisión, o incluso pueden compartir varias primitivas básicas (para representar por ejemplo cada parte de la articulación de un brazo robótico).

El **Mundo Físico** sobre el que se ejecuta el SDC mantiene una lista de todas las entidades que pueden colisionar empleando habitualmente una estructura global *Singleton*. Este *Mundo Físico* es una representación del mundo del juego que mantiene la información necesaria para la detección de las colisiones. Esta separación evita que el SDC tenga que acceder a estructuras de datos que no son necesarias para el cálculo de la colisión.

Los SDC mantienen estructuras de datos específicas para manejar las colisiones, proporcionando información sobre la *naturaleza* del contacto, que contiene la lista de las formas que están intersectando, su velocidad, etc...

Para gestionar de un modo más eficiente las colisiones, las formas que suelen utilizarse con convexas. Una **forma convexa** es aquella en la que un rayo que surja desde su interior atravesará la superfice una única vez. Las superficies convexas son mucho más simples y requieren menor capacidad computacional para calcular colisiones que las formas cóncavas. Algunas de las primitivas soportadas habitualmente en SDC son:

- **Esferas**. Son las primitivas más simples y eficientes; basta con definir su centro y radio (uso de un vector de 4 elementos).
- **Cajas**. Por cuestiones de eficiencia, se suelen emplear cajas límite alineadas con los ejes del sistema de coordenadas (AABB o ***Axis Aligned Bounding Box***). Las cajas AABB se definen mediante las coordenadas de dos extremos opuestos. El principal problema de las cajas AABB es que, para resultar eficientes, requieren estar alineadas con los ejes del sistema de coordenas global. Esto implica

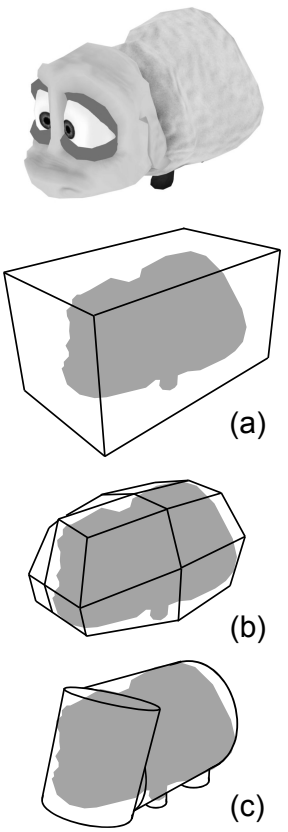


Figura 19.10: Diferentes formas de colisión para el objeto de la imagen. (a) Aproximación mediante una caja. (b) Aproximación mediante un volumen convexo. (c) Aproximación basada en la combinación de varias primitivas de tipo cilíndrico.

Formas de colisión

A cada cuerpo dinámico se le asocia habitualmente una única forma de colisión en el SDC.

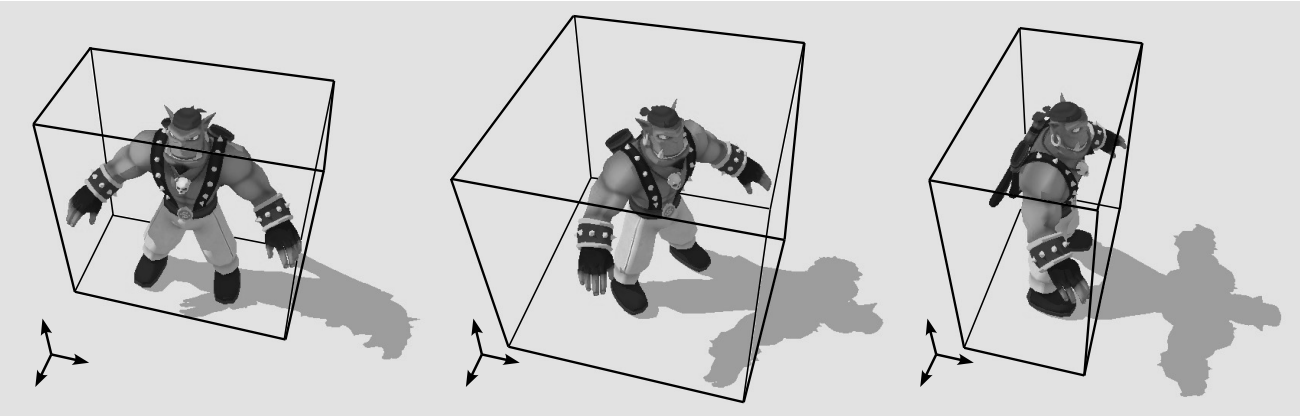


Figura 19.11: Gestión de la forma de un objeto empleando cajas límite alineadas con el sistema de referencia universal AABBs. Como se muestra en la figura, la caja central realiza una aproximación de la forma del objeto muy pobre.

que si el objeto rota, como se muestra en la Figura 19.11, la aproximación de forma puede resultar de baja calidad. Por su eficiencia, este tipo de cajas suelen emplearse para realizar una primera aproximación a la intersección de objetos para, posteriormente, emplear formas más precisas en el cálculo de la colisión.

Por su parte, las cajas OBB (***Oriented Bounding Box***) definen una rotación relativa al sistema de coordenadas. Su descripción es muy simple y permiten calcular la colisión entre primitivas de una forma muy eficiente.

- **Cilindros**. Los cilindros son ampliamente utilizados. Se definen mediante dos puntos y un radio. Una extensión de esta forma básica es la *cápsula*, que es un cuerpo compuesto por un cilindro y dos semiesferas (ver Figura 19.12). Puede igualmente verse como el volumen resultante de desplazar una esfera entre dos puntos. El cálculo de la intersección con cápsulas es más eficiente que con esferas o cajas, por lo que se emplean para el modelo de formas de colisión en formas que son aproximadamente cilíndricas (como las extremidades del cuerpo humano).

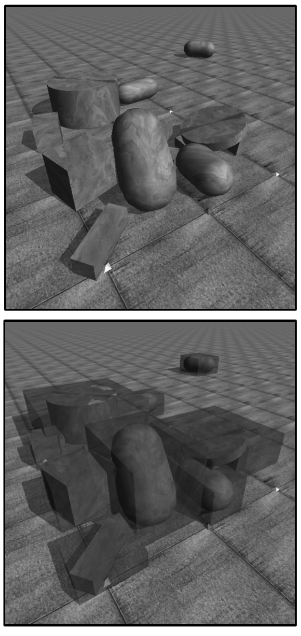


Figura 19.12: Algunas primitivas de colisión soportadas en ODE: Cajas, cilindros y cápsulas. La imagen inferior muestra los AABBs asociados a los objetos.

- **Volúmenes convexos**. La mayoría de los SDC permiten trabajar con volúmenes convexos (ver Figura 19.10). La forma del objeto suele representarse internamente mediante un conjunto de n planos. Aunque este tipo de formas es menos eficiente que las primitivas estudiadas anteriormente, existen ciertos algoritmos como el GJK que permiten optimizar los cálculos en este tipo de formas.
- **Malla poligional**. En ciertas ocasiones puede ser interesante utilizar mallas arbitrarias. Este tipo de superficies pueden ser abiertas (no es necesario que definan un volumen), y se construyen como mallas de triángulos. Las mallas poligonales se suelen emplear en elementos de geometría estática, como elementos del escenario, terrenos, etc. (ver Figura 19.13) Este tipo de formas de colisión son las más complejas computacionalmente, ya que el SDC debe probar con cada triángulo. Así, muchos juegos tratan de limitar el uso de este tipo de formas de colisión para evitar que el rendimiento se desplome.
- **Formas compuestas**. Este tipo de formas se utilizan cuando la descripción de un objeto se aproxima más convenientemente con una colección de formas. Este tipo de formas es la aproximación deseable en el caso de que tengamos que utilizar objetos cóncavos, que no se adaptan adecuadamente a volúmenes convexos (ver Figura 19.14).

19.2.2. Optimizaciones

La detección de colisiones es, en general, una tarea que requiere el uso intensivo de la CPU. Por un lado, los cálculos necesarios para determianr si dos formas intersecan no son triviales. Por otro lado, muchos juegos requiren un alto número de objetos en la escena, de modo que el número de test de intersección a calcular rápidamente crece. En el caso de n objetos, si empleamos un algoritmo de fuerza bruta tendríamos una complejidad $O(n^2)$. Es posible utilizar ciertos tipos de optimizaciones que mejoran esta complejidad inicial:

- **Coherencia Temporal**. Este tipo de técnica de optimización (también llamada *coherencia entre frames*), evita recalcular cierto tipo de información en cada frame, ya que entre pequeños intervalos de tiempo los objetos mantienen las posiciones y orientaciones en valores muy similares.
- **Particionamiento Espacial**. El uso de estructuras de datos de particionamiento especial permite comprobar rápidamente si dos objetos podrían estar intersecando si comparten la misma celda de la estructura de datos. Algunos esquemas de particionamiento jerárquico, como árboles octales, BSPs o árboles-kd permiten optimizar la detección de colisiones en el espacio. Estos esquemas tienen en común que el esquema de particionamiento comienza realizando una subdivisión general en la raíz, llegando a divisiones más finas y específicas en las hojas. Los objetos que se encuentran en una determinada rama de la estructura no pueden estar colisionando con los objetos que se encuentran en otra rama distinta.
- **Barrido y Poda (SAP)**. En la mayoría de los motores de simulación física se emplea un algoritmo Barrido y Poda (*Sweep and Prune*). Esta técnica ordena las cajas AABBs de los objetos de la escena y comprueba si hay intersecciones entre ellos. El algoritmo *Sweep and Prune* hace uso de la *Coherencia temporal frame a frame* para reducir la etapa de ordenación de $O(n \times log(n))$ a $O(n)$.

En muchos motores, como en Bullet, se utilizan varias capas o pasadas para detectar las colisiones. Primero suelen emplearse cajas AABB para comprobar si los objetos pueden estar potencialmente en colisión (detección de la colisión amplia). A continuación, en una segunda capa se hacen pruebas con volúmenes generales que engloban los objetos (por ejemplo, en un objeto compuesto por varios subobjetos, se calcula una esfera que agrupe a todos los subobjetos). Si esta segunda capa de colisión da un resultado positivo, en una tercera pasada se calculan las colisiones empleando las formas finales.

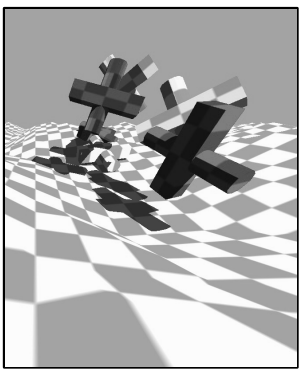


Figura 19.13: Bullet soporta la definición de mallas poligonales animadas. En este ejemplo de las demos oficiales, el suelo está animado y los objetos convexos colisionan respondiendo a su movimiento.

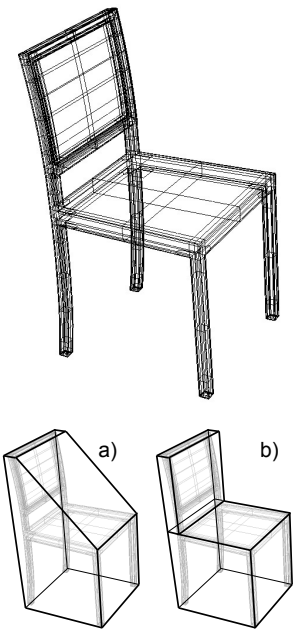


Figura 19.14: El modelo de una silla es un objeto que no se adapta bien a un volumen convexo (a). En (b) se ha utilizado una forma compuesta definiendo dos cajas.

19.2.3. Preguntando al sistema...

En el módulo 2 ya estudiamos algunas de las funcionalidades que se pueden encontrar en sistemas de detección de colisiones. El objetivo es poder obtener resultados a ciertas consultas sobre el primer objeto que intersecará con un determinado rayo, si hay objetos situados en el interior de un determinado volumen, etc.

A continuación veremos dos de las principales *collision queries* que pueden encontrarse habitualmente en un SDC:

- **Ray Casting**. Este tipo de *query* requiere que se especifique un rayo, y un origen. Si el rayo interseca con algún objeto, se devolverá un punto o una lista de puntos. Como vimos, el rayo se especifica habitualmente mediante una ecuación paramétrica de modo que $p(t) = p_o + td$, siendo t el parámetro que toma

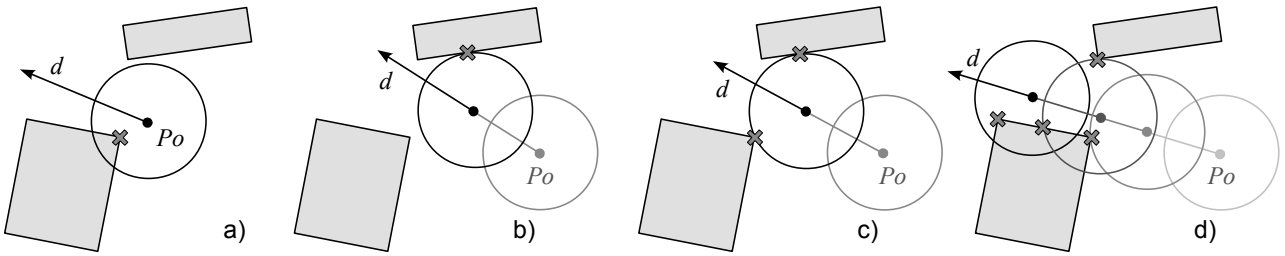


Figura 19.15: Cálculo de los puntos de colisión empleando *Shape Casting*. **a)** La forma inicialmente se encuentra colisionando con algún objeto de la escena. **b)** El SDC obtiene un punto de colisión. **c)** La forma interseca con dos objetos a la vez. **d)** En este caso el sistema calcula todos los puntos de colisión a lo largo de la trayectoria en la dirección de d.

valores entre 0 y 1. d nos define el vector dirección del rayo, que determinará la distancia máxima de cálculo de la colisión. P_o nos define el punto de origen del rayo. Este valor de t que nos devuelve la *query* puede ser fácilmente convertido al punto de colisión en el espacio 3D.

- **Shape Casting**. El *Shape Casting* permite determinar los puntos de colisión de una forma que viaja en la dirección de un vector determinado. Es similar al *Ray Casting*, pero en este caso es necesario tener en cuenta dos posibles situaciones que pueden ocurrir:

  1. La forma sobre la que aplicamos *Shape Casting* está inicialmente intersecando con al menos un objeto que evita que se desplace desde su posición inicial. En este caso el SDC devolverá los puntos de contacto que pueden estar situados sobre la superficie o en el interior del volumen.
  2. La forma no interseca sobre ningún objeto, por lo que puede desplazarse libremente por la escena. En este caso, el resultado de la colisión suele ser un punto de colisión situado a una determinada distancia del origen, aunque puede darse el caso de varias colisiones simultáneas (como se muestra en la Figura 19.15). En muchos casos, los SDC únicamente devuelven el resultado de la primera colisión (una lista de estructuras que contienen el valor de t, el identificador del objeto con el que han colisionado, el punto de contacto, el vector normal de la superficie en ese punto de contacto, y algunos campos extra de información adicional).

Uso de Rayos

El *Ray Casting* se utiliza ampliamente en videojuegos. Por ejemplo, para comprobar si un personaje está dentro del área de visión de otro personaje, para detectar si un disparo alcanza a un enemigo, para que los vehículos permanezcan en contacto con el suelo, etc.

Uso de Shape Casting

Un uso habitual de estas *queries* permite determinar si la cámara entra en colisión con los objetos de la escena, para ajustar la posición de los personajes en terrenos irregulares, etc.

Lista de colisiones

Para ciertas aplicaciones puede ser igualmente conveniente obtener la lista de todos los objetos con los que se interseca a lo largo de la trayectoria, como se muestra en la Figura 19.15.d).

19.3. Dinámica del Cuerpo Rígido

La simulación del movimiento de los objetos del juego forma parte del estudio de cómo las fuerzas afectan al comportamiento de los objetos. El módulo del motor de simulación que se encarga de la dinámica de los objetos estudia cómo cambian su posición en el tiempo. Hasta hace pocos años, los motores de simulación física se centraban en estudiar exclusivamente la *dinámica de cuerpos rígidos*[1], que permite simplificar el cálculo mediante dos suposiciones:

- Los objetos en la simulación obedecen las leyes del movimiento de Newton (estudiadas en la Sección 19.1.3). Así, no se tienen en cuenta ningún tipo de efecto cuántico ni relativista.

[1] Aunque en la actualidad se encuentran soportadas de una forma muy eficiente otras técnicas, como se muestran en la Figura 19.16

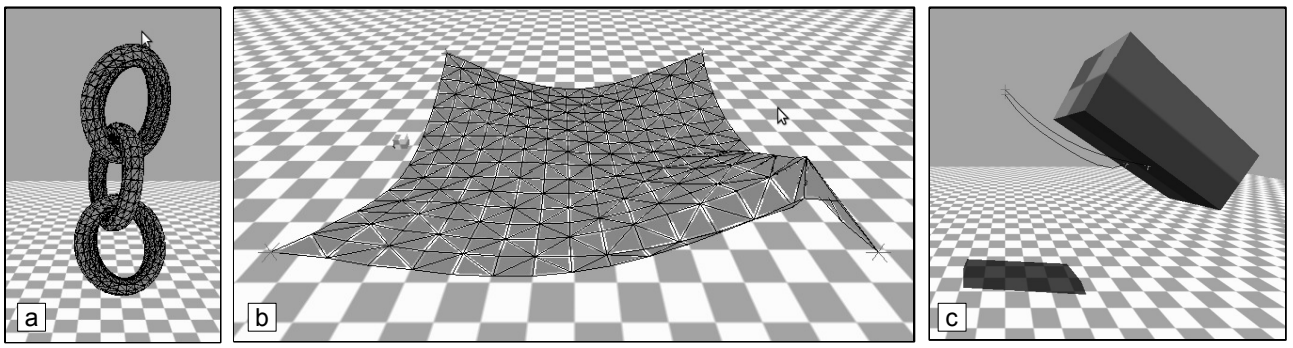


Figura 19.16: Ejemplo de simulación de cuerpos blandos con Bullet. a) Uso de *softbodies* con formas convexas. b) Simulación de una tela sostenida por cuatro puntos. c) Simulación de cuerdas.

- Todos los objetos que intervienen en la simulación son perfectamente sólidos y no se deforman. Esto equivale a afirmar que su forma es totalmente constante.

En el cálculo de la variación de la posición de los objetos con el tiempo, el motor de simulación necesita resolver ecuaciones diferenciales, que cuentan como variable independiente el tiempo. La resolución de estas ecuaciones habitualmente no puede realizarse de forma analítica (es imposible encontrar expresiones simples que relacionen las posiciones y velocidades de los objetos en función del tiempo), por lo que deben usarse métodos de integración numérica.

Gracias a los métodos de *integración numérica*, es posible resolver las ecuaciones diferenciales en pasos de tiempo, de modo que la solución en un instante de tiempo sirve como entrada para el siguiente paso de integración. La duración de cada *paso* de integración suele mantenerse constante Δt.

Uno de los métodos más sencillos que se pueden emplear es el de **Euler**, suponiendo que la velocidad del cuerpo es constante durante el incremento de tiempo. El método también presenta buenos resultados cuando Δt es suficientemente pequeño. En términos generales, este método no es suficientemente preciso, y tiene problemas de convergencia y de estabilidad (el sistema se vuelve inestable, no converge y hace que la simulación *explote*). La alternativa más utilizada en la actualidad es el método de integración de **Verlet**, por su bajo error y su eficiencia computacional en la evaluación de las expresiones.

19.4. Restricciones

Las restricciones sirven para limitar el movimiento de un objeto. Un objeto sin ninguna restricción tiene 6 grados de libertad. Las restricciones se usan en multitud de situaciones en desarrollo de videojuegos, como en puertas, suspensiones de vehículos, cadenas, cuerdas, etc. A continuación enumeraremos brevemente los principales tipos de restricciones soportadas por los motores de simulación física.

- **Punto a punto**. Este es el tipo de restricciones más sencillas; los objetos están conectados por un punto. Los objetos tienen libertad de movimiento salvo por el hecho de que tienen que mantenerse conectados por ese punto.

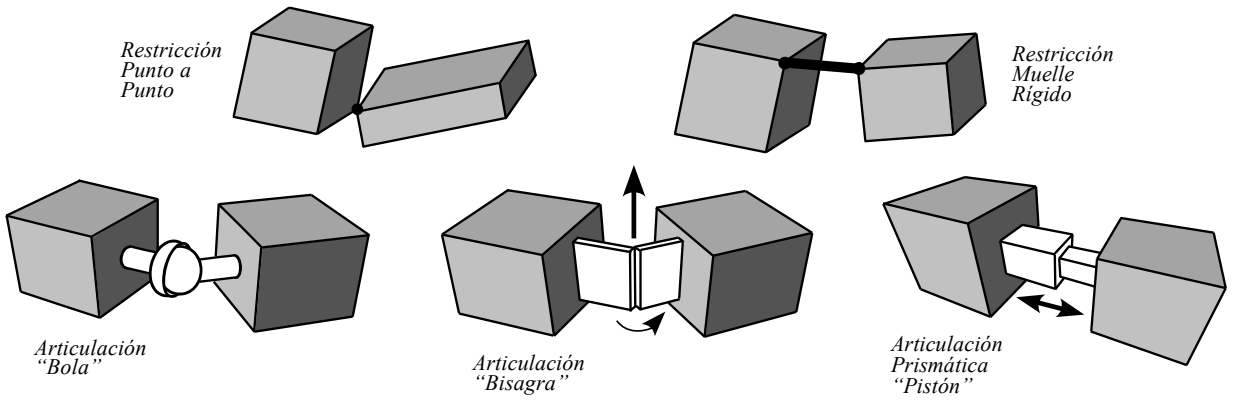


Figura 19.17: Representación de algunas de las principales restricciones que pueden encontrarse en los motores de simulación física.

- **Muelle rígido**. Un muelle rígido (*Stiff Spring*) funciona como una restricción de tipo punto a punto salvo por el hecho de que los objetos están separados por una determinada distancia. Así, puede verse como unidos por una *barra* rígida que los separa una distancia fija.
- **Bisagras**. Este tipo de restricción limitan el movimiento de rotación en un determinado eje (ver Figura 19.17). Este tipo de restricciones pueden definir además un límite de rotación angular permitido entre los objetos (para evitar, por ejemplo, que el codo de un brazo robótico adopte una posición imposible).
- **Pistones**. Este tipo de restricciones, denominadas en general *restricciones prismáticas*, permiten limitar el movimiento de traslación a un único eje. Estas restricciones podrían permitir opcionalmente rotación sobre ese eje.
- **Bolas**. Estas restricciones permiten definir límites de rotación flexibles, estableciendo puntos de anclaje. Sirven por ejemplo para modelar la rotación que ocurre en el hombro de un personaje.
- **Otras restricciones**. Cada motor permite una serie de restricciones específicas, como planares (que restringen el movimiento en un plano 2D), cuerdas, cadenas de objetos, *rag dolls* (definidas como una colección de cuerpos rígidos conectados utilizando una estructura jerárquica), etc...

19.5. Introducción a Bullet

Como se ha comentado al inicio del capítulo, Bullet es una biblioteca de simulación física muy utilizada, tanto en la industria del videojuego, como en la síntesis de imagen realista. Algunas de sus características principales son:

- Está desarrollada íntegramente en C++, y ha sido diseñada de modo que tenga el menor número de dependencias externas posibles.
- Se distribuye bajo licencia Zlib (licencia libre compatible con GPL), y ha sido utilizada en proyectos profesionales en multitud de plataformas, entre las que destacan PlayStation 3, XBox 360, Wii, Linux, Windows, MacOSX, iPhone y Android.

- Cuenta con un integrador muy estable y rápido. Permite el cálculo de dinámica de cuerpo rígido, dinámicas de vehículos y diversos tipos de restricciones (bisagras, pistones, bolas, etc...).
- Permite dinámica de *SoftBodies*, como telas, cuerdas y deformación de volúmenes arbitrarios.
- Las últimas versiones permiten descargar algunos cálculos a la GPU, empleando OpenCL. La utilización de esta funcionalidad se encuentra en fase experimental, y requiere la instalación de versiones recientes de los drivers de la tarjeta gráfica.
- Está integrado en multitud de paquetes de síntesis de imagen realista (bien de forma interna o mediante plugins), como en Blender, Maya, Softimage, Houdini, Cinema4D, etc...
- Permite importar y exportar archivos Collada.

Existen multitud de videojuegos comerciales que han utilizado Bullet como motor de simulación física. Entre otros se pueden destacar *Grand Theft Auto IV* (de Red Dead Redemption), *Free Realms* (de Sony), *HotWheels* (de BattleForce), *Blood Drive* (de Activision) o *Toy Story 3 (The Game)* (de Disney).

De igual forma, el motor se ha utilizado en películas profesionales, como *Hancock* (de Sony Pictures), *Bolt* de Walt Disney, *Sherlock Holmes* (de Framestore) o *Shrek 4* (de DreamWorks).

Muchos motores gráficos y de videojuegos permiten utilizar Bullet. La comunidad ha desarrollado multitud de *wrappers* para facilitar la integración en *Ogre*, *Crystal Space*, *Irrlich* y *Blitz3D* entre otros.

En el diseño de Bullet se prestó especial atención en conseguir un motor fácilmente adaptable y modular. Tal y como se comenta en su manual de usuario, el desarrollador puede utilizar únicamente aquellos módulos que necesite (ver Figura 19.19):

- Utilización exclusiva del componente de detección de colisiones.
- Utilización del componente de dinámicas de cuerpo rígido sin emplear los componentes de *SoftBody*.
- Utilización de pequeños fragmentos de código de la biblioteca.
- Extensión de la biblioteca para las necesidades específicas de un proyecto.
- Elección entre utilizar precisión simple o doble, etc...

Figura 19.18: Multitud de videojuegos AAA utilizan Bullet como motor de simulación física.

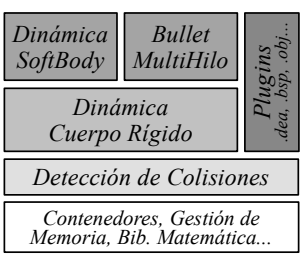


Figura 19.19: Esquema de los principales módulos funcionales de Bullet.

19.5.1. Pipeline de Físicas de Cuerpo Rígido

Bullet define el pipeline de procesamiento físico como se muestra en la Figura 19.20. El pipeline se ejecuta desde la izquierda a la derecha, comenzando por la etapa de cálculo de la gravedad, y finalizando con la integración de las posiciones (actualizando la transformación del mundo). Cada vez que se calcula un paso de la simulación `stepSimulation` en el mundo, se ejecutan las 7 etapas definidas en la imagen anterior.

El Pipeline comienza **aplicando la fuerza gravedad** a los objetos de la escena. Posteriormente se realiza una **predicción de las transformaciones** calculando la posición actual de los objetos. Hecho esto se **calculan las cajas AABBs**, que darán una estimación rápida de la posición de los objetos. Con estas cajas se **determinan los pares**, que consiste en calcular si las cajas AABBs se solapan en algún eje. Si no hay

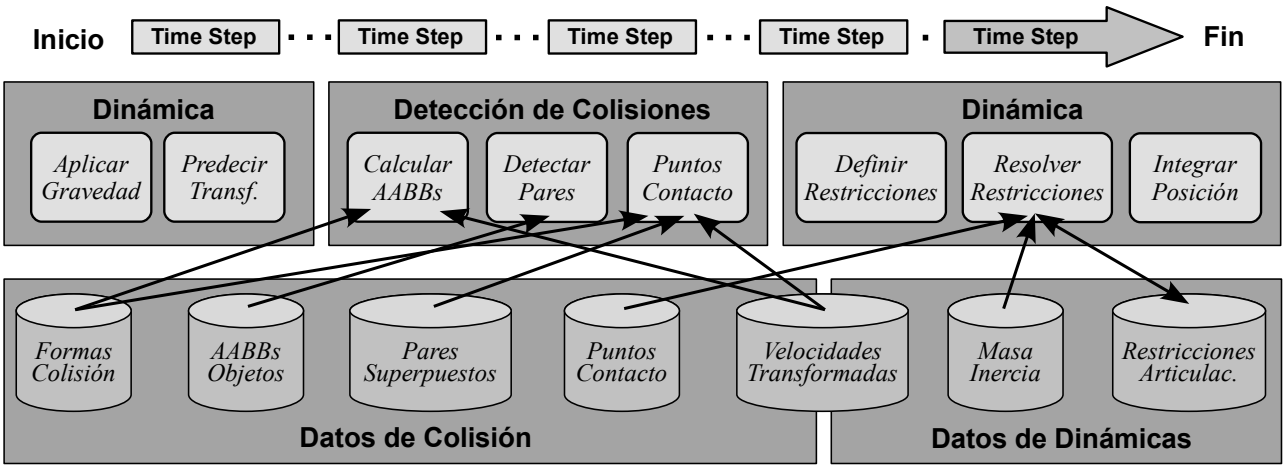


Figura 19.20: Pipeline de Bullet. En la parte superior de la imagen se describen las principales etapas computacionales, y en la parte inferior las estructuras de datos más importantes.

ningún solape, podemos afirmar que no hay colisión. De otra forma, hay que realizar un estudio más detallado del caso. Si hay posibilidad de contacto, se pasa a la siguiente etapa de **calcular los contactos**, que calcula utilizando la forma de colisión real del objeto el punto de contacto exacto. Estos puntos de contacto se pasan a la última etapa de cálculo de la dinámica, que comienza con la **resolución de las restricciones**, donde se determina la respuesta a la colisión empleando las restricciones de movimientos que se han definido en la escena. Finalmente se **integran las posiciones** de los objetos, obteniendo las posiciones y orientaciones nuevas para este paso de simulación.

Tipos básicos

Bullet cuenta con un subconjunto de utilidades matemáticas básicas con tipos de datos y operadores definidos como *btScalar* (escalar en punto flotante), *btVector3* (vector en el espacio 3D), *btTransform* (transformación afín 3D), *btQuaternion*, *btMatrix3x3*...

Las estructuras de datos básicas que define Bullet se dividen en dos grupos: los **datos de colisión**, que dependen únicamente de la *forma del objeto* (no se tiene en cuenta las propiedades físcas, como la masa o la velocidad asociada al objeto), y los **datos de propiedades dinámicas** que almacenan las propiedades físicas como la masa, la inercia, las restricciones y las articulaciones.

En los *datos de colisión* se distinguen las **formas de colisión**, las cajas *AABBs*, los **pares superpuestos** que mantiene una lista de parejas de cajas **AABB** que se solapan en algún eje, y los **puntos de contacto** que han sido calculados como resultado de la colisión.

19.5.2. Hola Mundo en Bullet

Para comenzar, estudiaremos el "Hola Mundo" en Bullet, que definirá una esfera que cae sobre un plano. Primero instalaremos las bibliotecas, que pueden descargarse de la página web del proyecto[2]. Desde el directorio donde tengamos el código descomprimido, ejecutamos:

```
cmake . -G "Unix Makefiles" -DINSTALL_LIBS=ON
make
sudo make install
```

Obviamente, es necesario tener instalado *cmake* para compilar la biblioteca. A continuación estudiaremos el código de la simulación.

[2] http://code.google.com/p/bullet/

Listado 19.1: Hello World en Bullet.

```
#include <iostream>
#include <btBulletDynamicsCommon.h>

int main (void) {
  btBroadphaseInterface* broadphase = new btDbvtBroadphase();
  btDefaultCollisionConfiguration* collisionConfiguration =
    new btDefaultCollisionConfiguration();
  btCollisionDispatcher* dispatcher = new btCollisionDispatcher(
      collisionConfiguration);
  btSequentialImpulseConstraintSolver* solver = new
      btSequentialImpulseConstraintSolver;
  btDiscreteDynamicsWorld* dynamicsWorld = new
      btDiscreteDynamicsWorld(dispatcher,broadphase,solver,
      collisionConfiguration);

  // Definicion de las propiedades del mundo ----------------------
  dynamicsWorld->setGravity(btVector3(0,-10,0));

  // Creacion de las formas de colision ---------------------------
  btCollisionShape* groundShape =
    new btStaticPlaneShape(btVector3(0,1,0),1);
  btCollisionShape* fallShape = new btSphereShape(1);

  // Definicion de los cuerpos rigidos en la escena ---------------
  btDefaultMotionState* groundMotionState = new
      btDefaultMotionState(btTransform(btQuaternion(0,0,0,1),
      btVector3(0,-1,0)));
  btRigidBody::btRigidBodyConstructionInfo
    groundRigidBodyCI(0,groundMotionState,groundShape,btVector3
        (0,0,0));
  btRigidBody* gRigidBody = new btRigidBody(groundRigidBodyCI);
  dynamicsWorld->addRigidBody(gRigidBody);

  btDefaultMotionState* fallMotionState =
    new btDefaultMotionState(btTransform(btQuaternion(0,0,0,1),
        btVector3(0,50,0)));
  btScalar mass = 1;
  btVector3 fallInertia(0,0,0);
  fallShape->calculateLocalInertia(mass,fallInertia);
  btRigidBody::btRigidBodyConstructionInfo fallRigidBodyCI(mass,
      fallMotionState,fallShape,fallInertia);
  btRigidBody* fallRigidBody = new btRigidBody(fallRigidBodyCI);
  dynamicsWorld->addRigidBody(fallRigidBody);

  // Bucle principal de la simulacion -----------------------------
  for (int i=0 ; i<300 ; i++) {
    dynamicsWorld->stepSimulation(1/60.f,10);    btTransform trans;
    fallRigidBody->getMotionState()->getWorldTransform(trans);
    std::cout << "Altura: " << trans.getOrigin().getY() << std::
        endl;
  }

  // Finalizacion (limpieza) --------------------------------------
  dynamicsWorld->removeRigidBody(fallRigidBody);
  delete fallRigidBody->getMotionState();  delete fallRigidBody;
  dynamicsWorld->removeRigidBody(gRigidBody);
  delete gRigidBody->getMotionState();  delete gRigidBody;
  delete fallShape;      delete groundShape;
  delete dynamicsWorld;  delete solver;
  delete collisionConfiguration;
  delete dispatcher;     delete broadphase;

  return 0;
}
```

El ejemplo anterior podría compilarse con un sencillo makefile como se muestra a continuación:

Listado 19.2: Makefile para hello world.

```
1 CXXFLAGS := `pkg-config --cflags bullet`
2 LDLIBS := `pkg-config --libs-only-l bullet`
3
4 all: HelloWorldBullet
5
6 clean:
7     rm HelloWorldBullet *~
```

```
Altura: 49.9972
Altura: 49.9917
Altura: 49.9833
Altura: 49.9722
Altura: 49.9583
Altura: 49.9417
Altura: 49.9222
Altura: 49.9
Altura: 49.875
Altura: 49.8472
Altura: 49.8167
Altura: 49.7833
Altura: 49.7472
Altura: 49.7083
Altura: 49.6667
...
```

Figura 19.21: Salida por pantalla de la ejecución del *Hola Mundo* en Bullet.

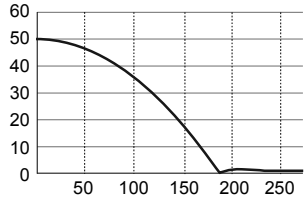


Figura 19.22: Representación de los valores obtenidos en el *Hola Mundo*.

Una vez compilado el programa de ejemplo, al ejecutarlo obtenemos un resultado puramente textual, como el mostrado en la Figura 19.21. Como hemos comentado anteriormente, los Bullet está diseñado para permitir un uso modular. En este primer ejemplo no se ha hecho uso de ninguna biblioteca para la representación gráfica de la escena.

Si representamos los 300 valores de la altura de la esfera, obtenemos una gráfica como muestra la Figura 19.22. Como vemos, cuando la esfera (de 1 metro de radio) llega a un metro del suelo (medido desde el centro), rebota levemente y a partir del frame 230 aproximadamente se estabiliza.

El primer `include` definido en la línea (2) del programa anterior se encarga de incluir todos los archivos de cabecera necesarios para crear una aplicación que haga uso del módulo de dinámicas (cuerpo rígido, restricciones, etc...). Necesitaremos instanciar un mundo sobre el que realizar la simulación. En nuestro caso crearemos un mundo discreto, que es el adecuado salvo que tengamos objetos de movimiento muy rápido sobre el que tengamos que hacer una detección de su movimiento (en cuyo caso podríamos utilizar la clase aún experimental *btContinuousDynamicsWorld*).

Creación del mundo

Como vimos en la sección 19.2.2, los motores de simulación física utilizan varias capas para detectar las colisiones entre los objetos del mundo. Bullet permite utilizar algoritmos en una primera etapa (*broadphase*) que utilizan las cajas límite de los objetos del mundo para obtener una lista de pares de cajas que pueden estar en colisión. Esta lista es una lista exhaustiva de todas las cajas límite que intersecan en alguno de los ejes (aunque, en etapas posteriores lleguen a descartarse porque en realidad no colisionan).

Muchos sistemas de simulación física se basan en el *Teorema de los Ejes Separados*. Este teorema dice que si existe un eje sobre el que la proyección de dos formas convexas no se solapan, entonces podemos asegurar que las dos formas no colisionan. Si no existe dicho eje y las dos formas son convexas, podemos asegurar que las formas colisionan. Si las formas son cóncavas, podría ocurrir que no colisionaran (dependiendo de la *suerte* que tengamos con la forma de los objetos). Este teorema se puede visualizar fácilmente en 2 dimensiones, como se muestra en la Figura 19.23.a.

El mismo teorema puede aplicarse para cajas AABB. Además, el hecho de que las cajas AABB estén perfectamente alineadas con los ejes del sistema de referencia, hace que el cálculo de la proyección de estas cajas sea muy rápida (simplemente podemos utilizar las coordenadas mínimas y máximas que definen las cajas).

La Figura 19.23 representa la aplicación de este teorema sobre cajas AABB. En el caso c) de dicha figura puede comprobarse que la proyección de las cajas se solapa en todos los ejes, por lo que tendríamos un caso potencial de colisión. En realidad, como vemos en la figura las formas que contienen dichas cajas AABB no colisionan, pero este caso deberá ser resuelto por algoritmos de detección de colisión de menor granularidad.

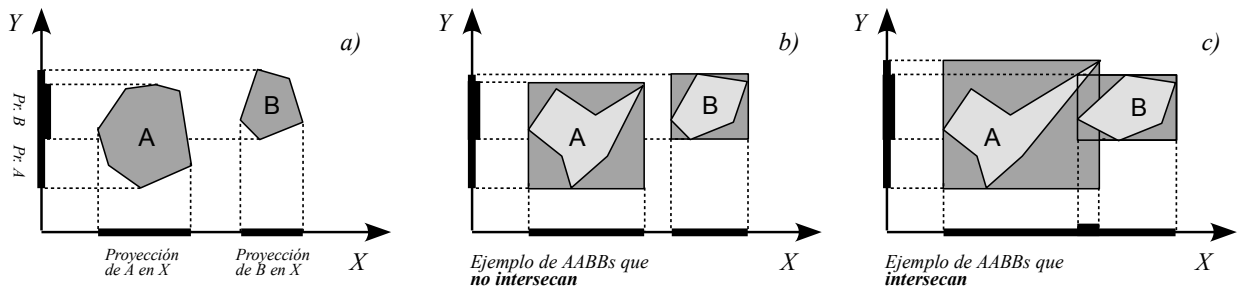


Figura 19.23: Aplicación del Teorema de los Ejes Separados y uso con cajas AABB. **a)** La proyección de las formas convexas de los objetos A y B están separadas en el eje X, pero no en el eje Y. Como podemos encontrar un eje sobre el que ambas proyecciones no intersecan, podemos asegurar que las formas no colisionan. **b)** El mismo principio aplicados sobre cajas AABB definidas en objetos cóncavos, que no intersecan. **c)** La proyección de las cajas AABB se solapa en todos los ejes. Hay un posible caso de colisión entre formas.

En la línea [5] creamos un objeto que implementa un algoritmo de optimización en una primera etapa *broadphase*. En posteriores etapas Bullet calculará las colisiones exactas. Existen dos algoritmos básicos que implementa Bullet para mejorar al aproximación *a ciegas* de complejidad $O(n^2)$ que comprobaría toda la lista de pares. Estos algoritmos añaden nuevas parejas de cajas que en realidad no colisionan, aunque en general mejoran el tiempo de ejecución.

- **Árbol AABB Dinámico**. Este algoritmo está implementado en la clase *btDbvtBroadphase*. Se construye un árbol AABB de propósito general que se utiliza tanto en la primera etapa de optimización *broadphase* como en la detección de colisiones entre *softbodies*. Este tipo de arbol se adapta automáticamente a las dimensiones del mundo, y la inserción y eliminación de objetos es más rápido que en SAP.
- **Barrido y Poda (*SAP*)**. La implementación de *Sweep and Prune* de Bullet requiere que el tamaño del mundo sea conocido de previamente. Este método es el que tiene mejor comportamiento en mundos dinámicos donde la mayoría de los objetos tienen poco movimiento. Se implementa en el conjunto de clases *AxisSweep* (con versiones de diverso nivel de precisión).

Tras esta primera *poda*, hemos eliminado gran cantidad de objetos que no colisionan. A continuación, en las líneas [6-8] se crea un objeto de configuración de la colisión, que nos permitirá adaptar los parámetros de los algoritmos utilizados en posteriores fases para comprobar la colisión. El *btCollisionDispatcher* es una clase que permite añadir funciones de *callback* para ciertos tipos de eventos (como por ejemplo, cuando los objetos se encuentren *cerca*).

El objeto *solver* (línea [9]) se encarga de que los objetos interactúen adecuadamente, teniendo en cuenta la gravedad, las fuerzas, colisiones y restricciones. En este ejemplo se ha utilizado la versión secuencial (que implementa el método de Gauss Seidel proyectado (PGS), para resolver problemas lineales), aunque existen versiones que hacen uso de paralelismo empleando hilos.

En la línea [10] se instancia el mundo. Este objeto nos permitirá añadir los objetos del mundo, aplicar gravedad, y avanzar el paso de la simulación. En concreto, en la línea [12] se establece una de las propiedades del mundo, la gravedad, asignando un valor de 10m/s en el eje Y, por lo que se aplicará sobre ese eje la fuerza de gravedad.

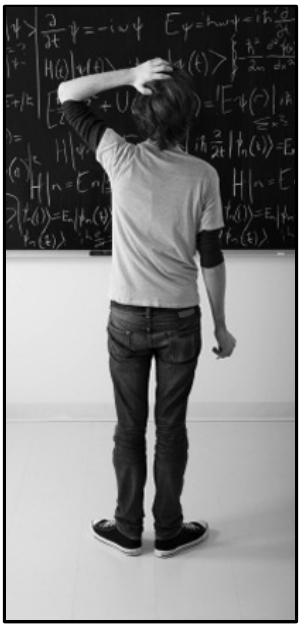


Figura 19.24: El objeto solver se encargará de resolver la interacción entre objetos.

Reutiliza!!

Es buena práctica reutilizar formas de colisión. Si varios objetos de la escena pueden compartir la misma forma de colisión (por ejemplo, todos los enemigos pueden gestionarse con una misma esfera de un determinado radio), es buena práctica compartir esa forma de colisión entre todos ellos.

Al finalizar, Bullet requiere que el usuario libere la memoria que ha utilizado explícitamente. De esta forma, a partir de la línea 42 se eliminan todos los elementos que han sido creados a lo largo de la simulación.

Hasta aquí hemos definido lo que puede ser el esqueleto básico de una aplicación mínima de Bullet. Vamos a definir a continuación los objetos que forman nuestra escena y el bucle de simulación.

Formas de Colisión

Como hemos comentado al inicio de la sección, crearemos un objeto plano que servirá como suelo sobre el que dejaremos caer una esfera. Cada uno de estos cuerpos necesita una forma de colisión, que internamente únicamente se utiliza para calcular la colisión (no tiene propiedades de masa, inercia, etc...).

Las formas de colisión no tienen una posición en el mundo; se adjuntan a los cuerpos rígidos. La elección de la forma de colisión adecuada, además de mejorar el rendimiento de la simulación, ayuda a conseguir una simulación de calidad. Bullet permite el uso de **primitivas** (que implementan algoritmos de detección de colisiones muy optimizados) o **mallas poligonales**. Las primitivas soportadas por Bullet son:

- **btSphereShape**. Esfera; la primitiva más simple y rápida.
- **btBoxShape**. La caja puede tener cualquier relación de aspecto.
- **btCylinderShape**. Cilindro con cualquier relación de aspecto.
- **btCapsuleShape**. Cápsula con cualquier relación de aspecto.
- **btConeShape**. Los conos se definen con el vértice en el (0,0,0).
- **btMultiSphereShape**. Forma convexa especial definida como combinación de esferas.
- **btCompoundShape**. No es una primitiva básica en sí, sino que permite combinar formas de cualquier tipo (tanto primitivas como formas de colisión de tipo malla que veremos a continuación). Permite obtener formas compuestas, como la que se estudió en la Figura 19.14.

Las formas de colisión de tipo malla soportadas son:

- **btConvexHull**. Este es el tipo de forma de tipo malla más rápido. Se define como una nube de vértices que forman la forma convexa más pequeña posible. El número de vértices debe ser pequeño para que la forma funcione adecuadamente. El número de vértices puede reducirse empleando la utilidad proporcionada por la clase *btShapeHull*. Existe una versión similar a este tipo llamado **btConvexTriangleMeshShape**, que está formado por caras triangulares, aunque es deseable utilizar *btConvexHull* porque es mucho más eficiente.
- **btBvhTriangleMeshShape**. Malla triangular estática. Puede tener un número considerable de polígonos, ya que utiliza una jerarquía interna para calcular la colisión. Como la construcción de esta estructura de datos puede llevar tiempo, se recomienda serializar el árbol para cargarlo rápidamente. Bullet incorpora utilidades para la serialización y carga del árbol BVH.
- **btHeightfieldTerrainShape**. Malla poligonal estática optimizada descrita por un mapa de alturas.

- **btStaticPlaneShape**. Plano infinito estático. Se especifica mediante un vector de dirección y una distancia respecto del origen del sistema de coordenadas.

Algunos consejos sobre el uso de formas de colisión en Bullet:

- Trata de utilizar las formas de colisión más eficientes: esferas, cajas, cilindros y *ConvexHull*.
- Los objetos dinámicos deben tener una forma cerrada y definida por un volumen finito. Algunas formas de colisión como los planos o las *triangleMesh* no tienen un volumen finito, por lo que únicamente pueden ser usados como cuerpos estáticos.
- Reutiliza siempre que sea posible las formas de colisión.

En la línea `16-17` creamos una forma de colisión de tipo plano, pasando como parámetro el vector normal del plano (vector unitario en Y), y una distancia respecto del origen. Así, el plano de colisión queda definido por la ecuación $y = 1$.

De igual modo, la forma de colisión del cuerpo que dejaremos caer sobre el suelo será una esfera de radio 1 metro (línea `18`).

Una vez definidas las formas de colisión, las posicionaremos asociándolas a instancias de cuerpos rígidos. En la siguiente subsección añadiremos los cuerpos rígidos al mundo.

Cuerpos Rígidos

Para añadir cuerpos rígidos, necesitamos primero definir el concepto de **MotionState** en Bullet. Un *MotionState* es una abstracción proporcionada por Bullet para actualizar la posición de los objetos que serán dibujados en el *game loop*. Empleando *MotionStates*, Bullet se encargará de actualizar los objetos que serán representados por el motor gráfico. En la siguiente sección estudiaremos cómo trabajar con *MotionStates* en Ogre.

Gracias al uso de *MotionStates*, únicamente se actualiza la posición de los objetos que se han movido. Bullet se encarga además de la interpolación de movimientos, aislando al programador de esta tarea. Cuando se consulte la posición de un objeto, por defecto se devolverá la correspondiente al último paso de simulación calculado. Sin embargo, cada vez que el motor gráfico necesite redibujar la escena, Bullet se encargará de devolver la transformación interpolada.

MotionState propio

Para implementar nuestro propio *MotionState* basta con heredar de `btMotionState` y sobreescribir los métodos `getWorldTransform` y `setWorldTransform`.

Los *MotionStates* deben utilizarse en dos situaciones:

1. Cuando se crea un cuerpo. Bullet determina la posición inicial del cuerpo en el momento de su creación, y requiere una llamada al *MotionState*.
2. Cuando se quiera actualizar la posición del objeto.

Bullet proporciona un *MotionState* por defecto que podemos utilizar para instanciar cuerpos rígidos. Así, en la línea `21` se utiliza el *MotionState* por defecto especificando como rotación la identidad, y trasladando el origen -1 unidad en Y [3].

[3] Esta traslación se realiza a modo de ejemplo para compensar la traslación de 1 unidad cuando se creó la forma de colisión del plano. El resultado sería el mismo si en ambos parámetros se hubiera puesto 0.

En las líneas 22–23 se emplea la estructura *btRigidBodyConstructionInfo* para establecer la información para crear un cuerpo rígido.

Los componentes de la estructura *btRigidBodyConstructionInfo* se copian a la información del cuerpo cuando se llama al constructor. Si queremos crear un grupo de objetos con las mismas propiedades, puede crearse una única estructura de este tipo y pasarla al constructor de todos los cuerpos.

El primer parámetro es la masa del objeto. Estableciendo una masa igual a cero (primer parámetro), se crea un objeto estático (equivale a establecer una masa infinita, de modo que el objeto no se puede mover). El último parámetro es la inercia del suelo (que se establece igualmente a 0, por ser un objeto estático).

En la línea 24 creamos el objeto rígido a partir de la información almacenada en la estructura anterior, y lo añadimos al mundo en la línea 25.

La creación de la esfera sigue un patrón de código similar. En la línea 27 se crea el *MotionState* para el objeto que dejaremos caer, situado a 50 metros del suelo (línea 28).

En las líneas 29–31 se establecen las propieades del cuerpo; una masa de 1Kg y se llama a un método de *btCollisionShape* que nos calcula la inercia de una esfera a partir de su masa.

Bucle Principal

Para finalizar, el bucle principal se ejecuta en las líneas 36–41. El bucle se ejecuta 300 veces, llamando al paso de simulación con un intervalo de 60hz. En cada paso de la simulación se imprime la altura de la esfera sobre el suelo.

Como puede verse, la posición y la orientación del objeto dinámico se encapsulan en un objeto de tipo *btTransform*. Como se comentó anteriormente, esta información puede obtenerse a partir del *MotionState* asociado al *btRigidBody* a través de la estructura de inicialización *btRigidBodyConstructInfo*.

Tiempos!

Cuidado, ya que las funciones de cálculo de tiempo habitualmente devuelven los resultados en milisegundos. Bullet trabaja en segundos, por lo que ésta es una fuente habitual de errores.

El método para avanzar un paso en la simulación (línea 38) requiere dos parámetros. El primero describe la cantidad de tiempo que queremos avanzar la simulación. Bullet tiene un reloj interno que permite mantener constante esta actualización, de forma que sea independiente de la tasa de frames de la aplicación. El segundo parámetro es el número de subpasos que debe realizar bullet cada vez que se llama *stepSimulation*. Los tiempos se miden en segundos.

No olvides...

Cuando cambies el valor de los tiempos de simulación, recuerda calcular el número de subpasos de simulación para que la ecuación siga siendo correcta.

El primer parámetro debe ser siempre menor que el número de subpasos multiplicado por el tiempo fijo de cada paso $t_{Step} < maxSubStep \times t_{FixedStep}$.

Decrementando el tamaño de cada paso de simulación se está aumentado la resolución de la simulación física. De este modo, si en el juego hay objetos que "atraviesan" objetos (como paredes), es posible decrementar el *fixedTimeStep* para aumentar la resolución. Obviamente, cuando se aumenta la resolución al doble, se necesitará aumentar el número de *maxSubSteps* al doble, lo que requerirá aproximadamente el doble de tiempo de CPU para el mismo tiempo de simulación física.

Supongamos que queremos un tiempo fijo de simulación en cada paso de 60hz. En el mejor de los casos, nuestro videojuego tendrá una tasa de 120fps (120hz), y en el peor de los casos de 12fps. Así, t_{Step} en el primer caso será $1/120 = 0{,}0083$, y en el segundo $t_{Step} = 1/12 = 0{,}083$. Por su parte, el tiempo del paso fijo para la simulación sería $1/60 = 0{,}017$. Para que la expresión anterior se cumpla, en el primer caso el número de subpasos basta con 1 $0{,}0083 < 1 \times 0{,}017$. En el peor de los casos, necesitaremos que el número de pasos sea al menos de 5 para que se cumpla la expresión $0{,}083 < 5 \times 0{,}017$. Con estas condiciones tendríamos que establecer el número de subpasos a 5 para no *perder tiempo de simulación*

Cuando se especifica un valor de $maxSubSteps > 1$, Bullet interpolará el movimiento (y evitará al programador tener que realizar los cálculos). Si $maxSubSteps == 1$, no realizará interpolación.

19.6. Integración manual en Ogre

Como se ha estudiado en la sección anterior, los *MotionStates* se definen en Bullet para abstraer la representación de los *rigidBody* en el motor de dibujado. A continuación definiremos manualmente una clase `MyMotionState` que se encargará de la actualización de las entidades en Ogre.

La implementación de un *MotionState* propio debe heredar de la clase *btMotionState* de bullet, y sobreescribir los métodos *getWorldTransform* y *setWorldTransform* (por lo que se definen como virtuales). Ambos métodos toman como parámetro un objeto de la clase *btTransform*, que se utiliza para la representación interna de transformaciones de cuerpo rígido.

El siguiente listado muestra la declaración de la clase, que tiene dos variables miembro; el nodo asociado a ese *MotionState* (que tendremos que actualizar en *setWorldTransform*), y la propia transformación que devolveremos en *getWorldTransform* (línea 7).

btTransform

Las transformaciones de cuerpo rígido están formadas únicamente por traslaciones y rotaciones (sin escalado). Así, esta clase utiliza internamente un *btVector3* para la traslación y una matriz 3x3 para almacenar la rotación.

Listado 19.3: MyMotionState.h

```
1  #include <Ogre.h>
2  #include <btBulletDynamicsCommon.h>
3
4  class MyMotionState : public btMotionState {
5  protected:
6    Ogre::SceneNode* _visibleobj;
7    btTransform _pos;
8
9  public:
10   MyMotionState(const btTransform &initialpos,
11                 Ogre::SceneNode* node);
12   virtual ~MyMotionState();
13   void setNode(Ogre::SceneNode* node);
14   virtual void getWorldTransform(btTransform &worldTr) const;
15   virtual void setWorldTransform(const btTransform &worldTr);
16 };
```

La definición de la clase es directa. El siguiente listado muestra los métodos más importantes en su implementación (el destructor no tiene que eliminar el nodo; se encargará Ogre al liberar los recursos).

En las líneas [15-19] se define el método principal, que actualiza la posición y rotación del *SceneNode* en Ogre. Dado que Bullet y Ogre definen clases distintas para trabajar con Vectores y Cuaternios, es necesario obtener la rotación y posición del objeto por separado y asignarlo al nodo mediante las llamadas a *setOrientation* y *setPosition* (líneas [17] y [19]).

La llamada a *setWorldTransform* puede retornar en la línea [15] si no se ha establecido nodo en el constructor. Se habilita un método específico para establecer el nodo más adelante. Esto es interesante si se quieren añadir objetos a la simulación que no tengan representación gráfica.

Listado 19.4: MyMotionState.cpp

```
1  #include "MyMotionState.h"
2
3  MyMotionState::MyMotionState(const btTransform &initialpos,
4    Ogre::SceneNode *node) {
5    _visibleobj = node;  _pos = initialpos;
6  }
7
8  void MyMotionState::setNode(Ogre::SceneNode *node)
9    { _visibleobj = node; }
10
11 void MyMotionState::getWorldTransform (btTransform &worldTr) const
12   { worldTr = _pos; }
13
14 void MyMotionState::setWorldTransform(const btTransform &worldTr){
15   if(NULL == _visibleobj) return;      // Si no hay nodo, return
16   btQuaternion rot = worldTr.getRotation();
17   _visibleobj->setOrientation(rot.w(), rot.x(), rot.y(), rot.z());
18   btVector3 pos = worldTr.getOrigin();
19   _visibleobj->setPosition(pos.x(), pos.y(), pos.z());
20 }
```

Una vez creada la clase que utilizaremos para definir el *MotionState*, la utilizaremos en el "*Hola Mundo*" construido en la sección anterior para representar la simulación con el plano y la esfera. El resultado que tendremos se muestra en la Figura 19.25. Para la construcción del ejemplo emplearemos como esqueleto base el *FrameListener* del Módulo 2 del curso.

Figura 19.25: Fragmento del resultado de integración del "Hola Mundo" de Bullet en Ogre, empleando la clase de *MyMotionState* definida anteriormente.

Listado 19.5: MyFrameListener.cpp

```
1  #include "MyFrameListener.h"
2  #include "MyMotionState.h"
3
4  MyFrameListener::MyFrameListener(RenderWindow* win,
5      Camera* cam, OverlayManager *om, SceneManager *sm) {
6    // .... Omitida parte de la inicializacion
7    _broadphase = new btDbvtBroadphase();
8    _collisionConf = new btDefaultCollisionConfiguration();
9    _dispatcher = new btCollisionDispatcher(_collisionConf);
10   _solver = new btSequentialImpulseConstraintSolver;
11   _world = new btDiscreteDynamicsWorld(_dispatcher,_broadphase,
12                         _solver,_collisionConf);
13   _world->setGravity(btVector3(0,-10,0));
14   CreateInitialWorld();
15 }
16
17 MyFrameListener::~MyFrameListener() {
18   _world->removeRigidBody(_fallRigidBody);
19   delete _fallRigidBody->getMotionState();
20   delete _fallRigidBody;
21   // ... Omitida la eliminacion de los objetos
22 }
23
```

```
24 void MyFrameListener::CreateInitialWorld() {
25   // Creacion de la entidad y del SceneNode ----------------------
26   Plane plane1(Vector3::Vector3(0,1,0), 0);
27   MeshManager::getSingleton().createPlane("p1",
28     ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane1,
29     200, 200, 1, 1, true, 1, 20, 20, Vector3::UNIT_Z);
30   SceneNode* node = _sceneManager->createSceneNode("ground");
31   Entity* groundEnt = _sceneManager->createEntity("planeEnt","p1");
32   groundEnt->setMaterialName("Ground");
33   node->attachObject(groundEnt);
34   _sceneManager->getRootSceneNode()->addChild(node);
35
36   // Creamos las formas de colision ------------------------------
37   _groundShape = new btStaticPlaneShape(btVector3(0,1,0),1);
38   _fallShape = new btSphereShape(1);
39
40   // Creamos el plano --------------------------------------------
41   MyMotionState* groundMotionState = new MyMotionState(
42     btTransform(btQuaternion(0,0,0,1),btVector3(0,-1,0)), node);
43   btRigidBody::btRigidBodyConstructionInfo groundRigidBodyCI
44     (0,groundMotionState,_groundShape,btVector3(0,0,0));
45   _groundRigidBody = new btRigidBody(groundRigidBodyCI);
46   _world->addRigidBody(_groundRigidBody);
47
48   // Creamos la esfera -------------------------------------------
49   Entity *entity2= _sceneManager->createEntity("ball","ball.mesh");
50   SceneNode *node2= _sceneManager->getRootSceneNode()->
       createChildSceneNode();
51   node2->attachObject(entity2);
52   MyMotionState* fallMotionState = new MyMotionState(
53      btTransform(btQuaternion(0,0,0,1),btVector3(0,50,0)), node2);
54   btScalar mass = 1;    btVector3 fallInertia(0,0,0);
55   _fallShape->calculateLocalInertia(mass,fallInertia);
56   btRigidBody::btRigidBodyConstructionInfo fallRigidBodyCI(
57      mass,fallMotionState,_fallShape,fallInertia);
58   _fallRigidBody = new btRigidBody(fallRigidBodyCI);
59   _world->addRigidBody(_fallRigidBody);
60 }
61
62 bool MyFrameListener::frameStarted(const Ogre::FrameEvent& evt) {
63   Real deltaT = evt.timeSinceLastFrame;
64   int fps = 1.0 / deltaT;
65
66   _world->stepSimulation(deltaT, 5);    // Actualizar fisica
67
68   _keyboard->capture();
69   if (_keyboard->isKeyDown(OIS::KC_ESCAPE)) return false;
70
71   btVector3 impulse;
72   if (_keyboard->isKeyDown(OIS::KC_I)) impulse=btVector3(0,0,-.1);
73   if (_keyboard->isKeyDown(OIS::KC_J)) impulse=btVector3(-.1,0,0);
74   if (_keyboard->isKeyDown(OIS::KC_K)) impulse=btVector3(0,0,.1);
75   if (_keyboard->isKeyDown(OIS::KC_L)) impulse=btVector3(.1,0,0);
76   _fallRigidBody->applyCentralImpulse(impulse);
77
78   // Omitida parte del codigo fuente (manejo del raton, etc...)
79   return true;
80 }
81
82 bool MyFrameListener::frameEnded(const Ogre::FrameEvent& evt) {
83   Real deltaT = evt.timeSinceLastFrame;
84   _world->stepSimulation(deltaT, 5);    // Actualizar fisica
85   return true;
86 }
```

En la implementación del *FrameListener* es necesario mantener como variables miembro el conjunto de objetos necesarios en la simulación de Bullet. Así, la implementación del constructor (líneas 4–15) define los objetos necesarios para crear el mundo de simulación de Bullet (línea 11–12). Estos objetos serán liberados en el destructor de la clase (ver líneas 17–22). De igual modo, los dos cuerpos rígidos que intervienen en la simulación y sus formas asociadas son variables miembro de la clase.

Variables miembro

Mantener los objetos como variables miembro de la clase no deja de ser una mala decisión de diseño. En la sección 19.7 veremos cómo se gestionan listas dinámicas con los objetos y las formas de colisión.

El método *CreateInitialWorld* (definido en las líneas 24–60) se realiza como último paso en el constructor. En este método se añaden a la escena de Ogre y al mundo de Bullet los elementos que intervendrán en la simulación (en este caso la esfera y el plano).

La creación de las entidades y los nodos para el plano y la esfera (líneas 26–34 y 49–51 respectivamente) ya han sido estudiadas en el Módulo 2 del curso. La creación de las formas para el plano y la esfera (líneas 37–38) fueron descritas en el código de la sección anterior. Cabe destacar que la malla exportada en `ball.mesh` (línea 49) debe tener un radio de 1 unidad, para que la forma de colisión definida en la línea 38 se adapte bien a su representación gráfica.

Cada objeto tendrá asociado un *MotionState* de la clase definida anteriormente, que recibirá la rotación y traslación inicial, y el puntero al nodo que guarda la entidad a representar. En el caso del plano, se define en las líneas 41–42, y la esfera en 52–53.

Por último, tendremos que añadir código específico en los métodos de retrollamada de actualización del frame. En el listado anterior se muestra el código de *frameStarted* (líneas 62–86). En la línea 66 se actualiza el paso de simulación de Bullet, empleando el tiempo transcurrido desde la última actualización. Además, si el usuario pulsa las teclas I, J, K o L (líneas 71–76), se aplicará una fuerza sobre la esfera. Veremos más detalles sobre la aplicación de impulsos a los objetos en el ejemplo de la sección 19.8.

Actualización del mundo

En el ejemplo anterior, se actualiza el paso de simulación igualmente en el método *frameEnded*. Bullet se encarga de interpolar las posiciones de dibujado de los objetos. Si se elimina la llamada a *stepSimulation*, el resultado de la simulación es mucho más brusco.

Para finalizar, se muestran los flags del Makefile necesarios para integrar Bullet en los ejemplos anteriores.

Listado 19.6: Fragmento de Makefile

```
1 # Flags de compilacion ------------------------------
2 CXXFLAGS := -I $(DIRHEA) -Wall `pkg-config --cflags OGRE` `pkg-
      config   --cflags bullet`
3
4 # Flags del linker ---------------------------------
5 LDFLAGS := `pkg-config --libs-only-L OGRE` `pkg-config --libs-only-
      l    bullet`
6 LDLIBS := `pkg-config --libs-only-l OGRE` `pkg-config --libs-only-l
       bullet` -lOIS -lGL -lstdc++
```

C19

19.7. Hola Mundo en OgreBullet

El desarrollo de un *wrapper* completo del motor Bullet puede ser una tarea costosa. Afortunadamente existen algunas alternativas que facilitan la integración del motor en Ogre, como el proyecto *OgreBullet*. OgreBullet se distribuye bajo una licencia MIT libre, y es multiplataforma.

Según el autor, *OgreBullet* puede ser considerado un wrapper en versión estable. La notificación de bugs, petición de nuevos requisitos y ejemplos se mantiene en un apartado específico de los foros de Ogre [4]. Uno de los principales problemas relativos al uso de este wrapper es la falta de documentación, por lo que en algunos casos la única alternativa es la consulta de los archivos de cabecera de la distribución.

OgreBullet puede descargarse de la página de complementos oficial de Ogre en: `http://ogreaddons.svn.sourceforge.net/viewvc/ogreaddons/trunk/ogrebullet/?view=tar`

En el siguiente ejemplo crearemos una escena donde se añadirán de forma dinámica cuerpos rígidos. Además de un puntero al *DynamicsWorld* (variable miembro *_world*), el *FrameListener* mantiene un puntero a un objeto *_debugDrawer* (línea 7), que nos permite representar cierta información visual que facilita el depurado de la aplicación. En este primer ejemplo se activa el dibujado de las formas de colisión (línea 8), tal y como se muestra en la Figura 19.26.

Este objeto permite añadir otros elementos que faciliten la depuración, como líneas, puntos de contacto, cajas AABBs, etc. Los métodos relativos a la representación de texto 3D en modo depuración están previstos pero aún no se encuentran desarrollados. Este objeto de depuración debe ser añadido igualmente al grafo de escena de Ogre (líneas 9-11 del siguiente listado).

La definición del mundo en *OgreBullet* requiere que se especifiquen los límites de simulación. En las líneas 14-15 se crea una caja AABB descrita por los vértices de sus esquinas que define el volumen en el que se realizará la simulación física. Este límite, junto con el vector de gravedad, permitirán crear el mundo (líneas 18-19).

Figura 19.26: Salida del primer ejemplo con *OgreBullet*. El objeto *_debugDrawer* muestra las formas de colisión asociadas a las entidades de Ogre.

Listado 19.7: Constructor

```
1  MyFrameListener::MyFrameListener(RenderWindow* win,
2      Camera* cam, OverlayManager *om, SceneManager *sm) {
3    _numEntities = 0;     // Numero de Formas instanciadas
4    _timeLastObject = 0; // Tiempo desde ultimo objeto anadido
5
6    // Creacion del modulo de debug visual de Bullet ---------------
7    _debugDrawer = new OgreBulletCollisions::DebugDrawer();
8    _debugDrawer->setDrawWireframe(true);
9    SceneNode *node = _sceneManager->getRootSceneNode()->
10     createChildSceneNode("debugNode", Vector3::ZERO);
11   node->attachObject(static_cast<SimpleRenderable*>(_debugDrawer));
12   // Creacion del mundo (definicion de los limites y la gravedad)
13   AxisAlignedBox worldBounds = AxisAlignedBox
14     (Vector3(-10000,-10000,-10000), Vector3(10000,10000,10000));
15   Vector3 gravity = Vector3(0, -9.8, 0);
16   _world = new OgreBulletDynamics::DynamicsWorld(_sceneManager,
17        worldBounds, gravity);
18   _world->setDebugDrawer (_debugDrawer);
19   _world->setShowDebugShapes (true);  // Muestra formas debug
20   CreateInitialWorld();    // Inicializa el mundo
21 }
```

[4] Foros OgreBullet: `http://www.ogre3d.org/addonforums/viewforum.php?f=12`

El *FrameListener* mantiene dos colas de doble fin (*deque*) de punteros a los *RigidBody* (_bodies) y a las *CollisionShape* (_shapes), que facilitan la inserción y borrado de elementos de un modo más rápido que los vectores. Así, cuando se añadan objetos de forma dinámica a la escena, será necesario añadirlos a estas estructuras para su posterior liberación en el destructor de la clase. El siguiente listado muestra la implementación del destructor del *FrameListener*.

De igual modo es necesario liberar los recursos asociados al mundo dinámico y al *debugDrawer* creado en el constructor (ver líneas 16–17).

Listado 19.8: Destructor

```
1  MyFrameListener::~MyFrameListener() {
2    // Eliminar cuerpos rigidos --------------------------------------
3    std::deque <OgreBulletDynamics::RigidBody *>::iterator
4       itBody = _bodies.begin();
5    while (_bodies.end() != itBody) {
6      delete *itBody;  ++itBody;
7    }
8    // Eliminar formas de colision ----------------------------------
9    std::deque<OgreBulletCollisions::CollisionShape *>::iterator
10     itShape = _shapes.begin();
11   while (_shapes.end() != itShape) {
12     delete *itShape; ++itShape;
13   }
14   _bodies.clear();  _shapes.clear();
15   // Eliminar mundo dinamico y debugDrawer ------------------------
16   delete _world->getDebugDrawer();    _world->setDebugDrawer(0);
17   delete _world;
18 }
```

Para añadir un objeto de simulación de *OgreBullet* debemos crear dos elementos básicos; por un lado la *CollisionShape* (líneas 14–16), y por otro lado el *RigidBody* (líneas 17–18).

La asociación del nodo de dibujado con el cuerpo rígido se establece en la misma llamada en la que se asocia la forma de colisión al *RigidBody*. En la clase *OgreBulletDynamics::RigidBody* existen dos métodos que permiten asociar una forma de colisión a un *RigidBody*; *setShape* y *setStaticShape*. La segunda cuenta con varias versiones; una de ellas no requiere especificar el *SceneNode*, y se corresponde con la utilizada en la línea 21 para añadir la forma de colisión al plano.

Añadir a las colas

Una vez añadido el objeto, deben añadirse las referencias a la forma de colisión y al cuerpo rígido en las colas de la clase (línea 24).

Listado 19.9: CreateInitialWorld

```
1  void MyFrameListener::CreateInitialWorld() {
2    // Creacion de la entidad y del SceneNode -----------------------
3    Plane plane1(Vector3::Vector3(0,1,0), 0);
4    MeshManager::getSingleton().createPlane("p1",
5      ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane1,
6      200, 200, 1, 1, true, 1, 20, 20, Vector3::UNIT_Z);
7    SceneNode* node= _sceneManager->createSceneNode("ground");
8    Entity* groundEnt= _sceneManager->createEntity("planeEnt", "p1");
9    groundEnt->setMaterialName("Ground");
10   node->attachObject(groundEnt);
11   _sceneManager->getRootSceneNode()->addChild(node);
12
13   // Creamos forma de colision para el plano ----------------------
14   OgreBulletCollisions::CollisionShape *Shape;
15   Shape = new OgreBulletCollisions::StaticPlaneCollisionShape
16     (Ogre::Vector3(0,1,0), 0);   // Vector normal y distancia
17   OgreBulletDynamics::RigidBody *rigidBodyPlane = new
18     OgreBulletDynamics::RigidBody("rigidBodyPlane", _world);
19
20   // Creamos la forma estatica (forma, Restitucion, Friccion) ----
```

```
21   rigidBodyPlane->setStaticShape(Shape, 0.1, 0.8);
22
23   // Anadimos los objetos Shape y RigidBody ----------------------
24   _shapes.push_back(Shape);      _bodies.push_back(rigidBodyPlane);
25 }
```

La actualización del mundo se realiza de forma similar a la estudiada anteriormente. La aplicación de ejemplo además, permite añadir objetos dinámicos cuando se pulse la tecla [B] (línea [9]). A continuación veremos el código para añadir cuerpos dinámicos.

stepSimulation

La llamada a *stepSimulation* en OgreBullet acepta dos parámetros opcionales, el número de subpasos de simulación (por defecto a 1), y el *fixedTimeStep* (por defecto 1/60).

Listado 19.10: FrameStarted

```
1  bool MyFrameListener::frameStarted(const Ogre::FrameEvent& evt) {
2    Ogre::Real deltaT = evt.timeSinceLastFrame;
3    _world->stepSimulation(deltaT); // Actualizar simulacion Bullet
4    _timeLastObject -= deltaT;
5
6    _keyboard->capture();
7    if (_keyboard->isKeyDown(OIS::KC_ESCAPE)) return false;
8    if ((_keyboard->isKeyDown(OIS::KC_B)) && (_timeLastObject <= 0))
9      AddDynamicObject();
10   // Omitido el resto del cogido del metodo ...
11   return true;
12 }
```

El siguiente listado implementa la funcionalidad de añadir cajas dinámicas a la escena. Los objetos se crearán teniendo en cuenta la posición y rotación de la cámara. Para ello, se toma como vector de posición inicial el calculado como la posición de la cámara desplazada 10 unidades según su vector dirección (líneas [5-6]).

Listado 19.11: AddDynamicObject

```
1  void MyFrameListener::AddDynamicObject() {
2    _timeLastObject = 0.25;   // Segundos para anadir uno nuevo...
3
4    Vector3 size = Vector3::ZERO;   // Tamano y posicion inicial
5    Vector3 position = (_camera->getDerivedPosition()
6       + _camera->getDerivedDirection().normalisedCopy() * 10);
7
8    // Creamos la entidad y el nodo de la escena --------------------
9    Entity *entity = _sceneManager->createEntity("Box" +
10      StringConverter::toString(_numEntities), "cube.mesh");
11   entity->setMaterialName("cube");
12   SceneNode *node = _sceneManager->getRootSceneNode()->
13     createChildSceneNode();
14   node->attachObject(entity);
15
16   // Obtenemos la bounding box de la entidad creada --------------
17   AxisAlignedBox boundingB = entity->getBoundingBox();
18   size = boundingB.getSize();
19   size /= 2.0f;   // Tamano en Bullet desde el centro (la mitad)
20   OgreBulletCollisions::BoxCollisionShape *boxShape = new
21     OgreBulletCollisions::BoxCollisionShape(size);
22   OgreBulletDynamics::RigidBody *rigidBox = new
23     OgreBulletDynamics::RigidBody("rigidBox" +
24        StringConverter::toString(_numEntities), _world);
25   rigidBox->setShape(node, boxShape,
26          /* Restitucion, Friccion, Masa */ 0.6, 0.6, 5.0,
27          /* Pos. y Orient. */ position , Quaternion::IDENTITY);
28   rigidBox->setLinearVelocity(
29      _camera->getDerivedDirection().normalisedCopy() * 7.0);
30   _numEntities++;
31   // Anadimos los objetos a las deques ---------------------------
32   _shapes.push_back(boxShape);   _bodies.push_back(rigidBox);
```

```
33 }
```

En el listado anterior, el nodo asociado a cada caja se añade en la llamada a *setShape* (líneas 25-27). Pese a que Bullet soporta multitud de propiedades en la estructura *btRigidBodyConstructionInfo*, el wrapper se centra exclusivamente en la definición de la masa, y los coeficientes de fricción y restitución. La posición inicial y el cuaternio se indican igualmente en la llamada al método, que nos abstrae de la necesidad de definir el *MotionState*.

Las cajas se añaden a la escena con una velocidad lineal relativa a la rotación de la cámara (ver líneas 28-29).

19.8. RayQueries

Al inicio del capítulo estudiamos algunos tipos de preguntas que podían realizarse al motor de simulación física. Uno de ellos eran los *RayQueries* que permitían obtener las formas de colisión que intersecaban con un determinado rayo.

Ogre? Bullet?

Recordemos que los objetos de simulación física no son conocidos por Ogre. Aunque en el módulo 2 del curso estudiamos los RayQueries en Ogre, es necesario realizar la pregunta en Bullet para obtener las referencias a los *RigidBody*.

Utilizaremos esta funcionalidad del SDC para aplicar un determinado impulso al primer objeto que sea *tocado* por el puntero del ratón. De igual forma, en este ejemplo se añadirán objetos definiendo una forma de colisión convexa. El resultado de la simulación (activando la representación de las formas de colisión) se muestra en la Figura 19.27.

La llamada al método *AddDynamicObject* recibe como parámetro un tipo enumerado, que indica si queremos añadir una oveja o una caja. La forma de colisión de la caja se calcula automáticamente empleando la clase *StaticMeshToShapeConverter* (línea 17).

Reutiliza las formas de colisión! Tanto en el ejemplo de la sección anterior como en este código, no se reutilizan las formas de colisión. Queda como ejercicio propuesto para el lector mantener referencias a las formas de colisión (para las ovejas y para las cajas), y comprobar la diferencia de rendimiento en frames por segundo cuando el número de objetos de la escena crece.

Listado 19.12: AddDynamicObject

```
1  void MyFrameListener::AddDynamicObject(TEDynamicObject tObject) {
2    // Omitido codigo anterior del metodo ---------------------------
3    Entity *entity = NULL;
4    switch (tObject) {
5    case sheep:
6       entity = _sceneManager->createEntity("Sheep" +
7       StringConverter::toString(_numEntities), "sheep.mesh");
8      break;
9    case box: default:
10      // (Omitido) Analogamente se carga el modelo de la caja...
11   }
12
13   SceneNode *node = _sceneManager->getRootSceneNode()->
14     createChildSceneNode();
15   node->attachObject(entity);
16
17   OgreBulletCollisions::StaticMeshToShapeConverter *
         trimeshConverter = NULL;
```

```
18   OgreBulletCollisions::CollisionShape *bodyShape = NULL;
19   OgreBulletDynamics::RigidBody *rigidBody = NULL;
20
21   switch (tObject) {
22   case sheep:
23     trimeshConverter = new
24       OgreBulletCollisions::StaticMeshToShapeConverter(entity);
25     bodyShape = trimeshConverter->createConvex();
26     delete trimeshConverter;
27     break;
28   case box: default:
29     // (Omitido) Crear bodyShape como en el ejemplo anterior...
30   }
31
32   rigidBody = new OgreBulletDynamics::RigidBody("rigidBody" +
33      StringConverter::toString(_numEntities), _world);
34   // Omitido resto de codigo del metodo ---------------------------
35 }
```

El objeto de la clase *StaticMeshToShapeConverter* recibe como parámetro una *Entity* de Ogre en el constructor. Esta entidad puede ser convertida a multitud de formas de colisión. En el momento de la creación, la clase reduce el número de vértices de la forma de colisión.

Cuando se pincha con el botón derecho o izquierdo del ratón sobre algún objeto de la simulación, se aplicará un impulso con diferente fuerza (definida en F, ver línea [18] del siguiente código). El método *pickBody* se encarga de obtener el primer cuerpo que colisiona con el rayo definido por la posición de la cámara y el puntero del ratón. Este método devuelve igualmente en los dos primeros parámetros el punto de colisión en el objeto y el rayo utilizado para construir el *RayQuery*.

El método *pickBody* primero obtiene el rayo utilizando la funcionalidad de Ogre, empleando las coordenadas de pantalla normalizadas (líneas [20-21]). Hecho esto, se crea una *Query* que requiere como tercer parámetro la distancia máxima a la que se calculará la colisión, en la dirección del rayo, teniendo en cuenta su posición inicial (línea [4]).

Si el rayo colisiona en algún cuerpo (línea [6]), se devuelve el cuerpo y el punto de colisión (líneas [7-11]).

Listado 19.13: RayQuery en Bullet

```
1  RigidBody* MyFrameListener::pickBody (Vector3 &p, Ray &r, float x,
       float y) {
2    r = _camera->getCameraToViewportRay (x, y);
3    CollisionClosestRayResultCallback cQuery =
4      CollisionClosestRayResultCallback (r, _world, 10000);
5    _world->launchRay(cQuery);
6    if (cQuery.doesCollide()) {
7      RigidBody* body = static_cast <RigidBody *>
8        (cQuery.getCollidedObject());
9      p = cQuery.getCollisionPoint();
10     return body;
11   }
12   return NULL;
13 }
14
15 bool MyFrameListener::frameStarted(const Ogre::FrameEvent& evt) {
16   // Omitido codigo anterior del metodo ---------------------------
17   if (mbleft || mbright) { // Con botones del raton, impulso ------
18     float F = 10;  if (mbright) F = 100;
19     RigidBody* body; Vector3 p; Ray r;
20     float x = posx/float(_win->getWidth());   // Pos x normalizada
21     float y = posy/float(_win->getHeight());  // Pos y normalizada
22     body = pickBody (p, r, x, y);
```

Figura 19.27: Resultado de la simulación del ejemplo.

Conversor a Shape

Además de la llamada a *createConvex*, el conversor estudiado en el código anterior puede generar otras formas de colisión con *createSphere*, *createBox*, *createTrimesh*, *createCylinder* y *createConvexDecomposition* entre otras.

```
23
24      if (body) {
25        if (!body->isStaticObject()) {
26            body->enableActiveState ();
27            Vector3 relPos(p - body->getCenterOfMassPosition());
28            Vector3 impulse (r.getDirection ());
29            body->applyImpulse (impulse * F, relPos);
30        }
31      }
32    }
33    // Omitido resto de codigo del metodo ----------------------------
34  }
```

Para finalizar, si hubo colisión con algún cuerpo que no sea estático (líneas 24-25), se aplicará un impulso. La llamada a *enableActiveState* permite activar un cuerpo. Por defecto, Bullet automáticamente desactiva objetos dinámicos cuando la velocidad es menor que un determinado umbral.

Los cuerpos desactivados en realidad están se encuentran en un estado de *dormidos*, y no consumen tiempo de ejecución salvo por la etapa de detección de colisión *broadphase*. Esta etapa automáticamente despierta a los objetos que estuvieran dormidos si se encuentra colisión con otros elementos de la escena.

Impulso

El impulso puede definirse como $\int Fdt = \int (dp/dt)dt$, siendo p el momento.

En las líneas 27-29 se aplica un impulso sobre el objeto en la dirección del rayo que se calculó desde la cámara, con una fuerza proporcional a F. El impulso es una fuerza que actúa en un cuerpo en un determinado intervalo de tiempo. El impulso implica un cambio en el *momento*, siendo la *Fuerza* definida como el cambio en el momento. Así, el *impulso* aplicado sobre un objeto puede ser definido como la integral de la fuerza con respecto del tiempo.

El wrapper de *OgreBullet* permite definir un pequeño subconjunto de propiedades de los *RigidBody* de las soportadas en Bullet. Algunas de las principales propiedades son la Velocidad Lineal, Impulsos y Fuerzas. Si se requieren otras propiedades, será necesario acceder al objeto de la clase *btRigidBody* (mediante la llamada a *getBulletRigidBody*) y especificar manualmente las propiedades de simulación.

19.9. TriangleMeshCollisionShape

En este ejemplo se cargan dos objetos como mallas triangulares estáticas. El resultado de la ejecución puede verse en la Figura 19.28. Al igual que en el ejemplo anterior, se utiliza la funcionalidad proporcionada por el conversor de mallas, pero generando una *TriangleMeshCollisionShape* (línea 11-12).

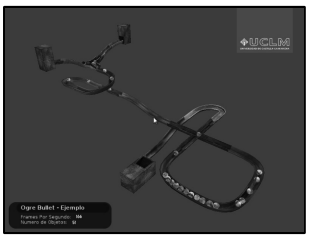


Figura 19.28: Resultado de la ejecución del ejemplo de carga de mallas triangulares.

Listado 19.14: Static Mesh

```
1  void MyFrameListener::CreateInitialWorld() {
2    // Creacion del track ----------------------------------------
3    Entity *entity = _sceneManager->createEntity("track.mesh");
4    SceneNode *node = _sceneManager->createSceneNode("track");
5    node->attachObject(entity);
6
7    _sceneManager->getRootSceneNode()->addChild(node);
8    OgreBulletCollisions::StaticMeshToShapeConverter *
         trimeshConverter = new
9      OgreBulletCollisions::StaticMeshToShapeConverter(entity);
10
11   OgreBulletCollisions::TriangleMeshCollisionShape *trackTrimesh =
12     trimeshConverter->createTrimesh();
13
14   OgreBulletDynamics::RigidBody *rigidTrack = new
15     OgreBulletDynamics::RigidBody("track", _world);
```

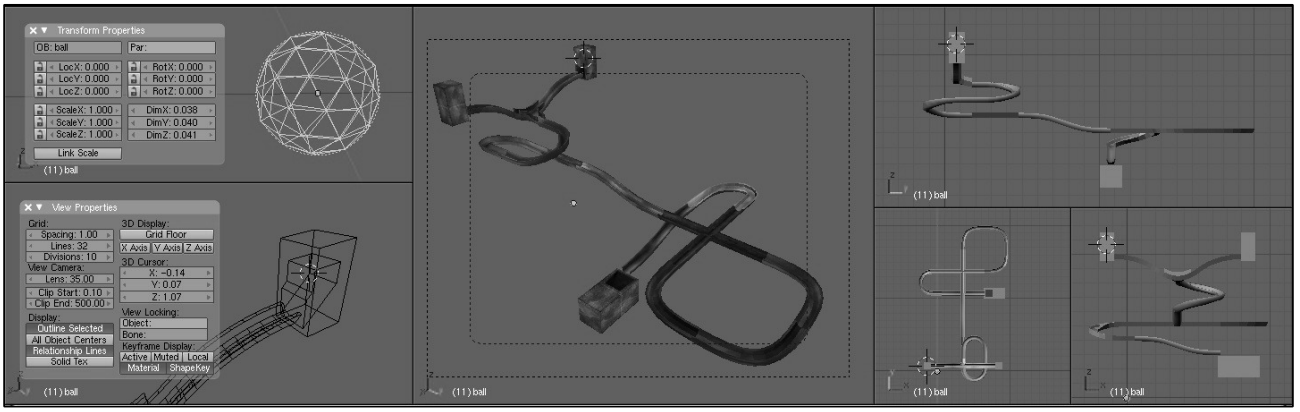


Figura 19.29: Configuración de la malla estática utilizada en el ejemplo. Es importante aplicar la escala y rotación a los objetos antes de su exportación, así como las dimensiones del objeto "ball" para aplicar los mismos límites a la *collision shape*.

```
16    rigidTrack->setShape(node, trackTrimesh, 0.8, 0.95, 0,
17      Vector3::ZERO, Quaternion::IDENTITY);
18
19    delete trimeshConverter;
20    // (Omitido) Creacion del sumidero de forma similar ------------
21  }
```

Es importante consultar en la posición de los generadores de objetos en el espacio 3D (ver Figura 19.29), así como las dimensiones de los objetos que van a intervenir en la simulación. Por ejemplo, las esferas se creaban con una forma de colisión de tipo SphereCollisionShape de 0.02 unidades de radio porque su dimensión en el espacio 3D es de 0.04 unidades (ver Figura 19.29). De igual modo, una de las posiciones de generación es *Vector3(-0.14, 1.07, -0.07)* situada en el interior de una de las cajas.

19.10. Detección de colisiones

Una de las formas más sencillas de detectar colisiones entre objetos del mundo es iterar sobre los colectores de contactos (*contact manifold*). Los *contact manifold* son en realidad caches que contienen los puntos de contacto entre parejas de objetos de colisión. El siguiente listado muestra una forma de iterar sobre los pares de objetos en el mundo dinámico.

Listado 19.15: DetectCollisionDrain.

```
1  void MyFrameListener::DetectCollisionDrain() {
2    btCollisionWorld *bulletWorld=_world->getBulletCollisionWorld();
3    int numManifolds=bulletWorld->getDispatcher()->getNumManifolds();
4
5    for (int i=0;i<numManifolds;i++) {
6      btPersistentManifold* contactManifold =
7        bulletWorld->getDispatcher()->getManifoldByIndexInternal(i);
8      btCollisionObject* obA =
9        static_cast<btCollisionObject*>(contactManifold->getBody0());
10     btCollisionObject* obB =
11       static_cast<btCollisionObject*>(contactManifold->getBody1());
12
13     Ogre::SceneNode* drain = _sceneManager->getSceneNode("drain");
14
```

```
15      OgreBulletCollisions::Object *obDrain =
16                                          _world->findObject(drain);
17      OgreBulletCollisions::Object *obOB_A = _world->findObject(obA);
18      OgreBulletCollisions::Object *obOB_B = _world->findObject(obB);
19
20      if ((obOB_A == obDrain) || (obOB_B == obDrain)) {
21        Ogre::SceneNode* node = NULL;
22        if ((obOB_A != obDrain) && (obOB_A)) {
23          node = obOB_A->getRootNode(); delete obOB_A;
24        }
25        else if ((obOB_B != obDrain) && (obOB_B)) {
26          node = obOB_B->getRootNode(); delete obOB_B;
27        }
28        if (node) {
29          std::cout << node->getName() << std::endl;
30          _sceneManager->getRootSceneNode()->
31            removeAndDestroyChild (node->getName());
32        }
33      }
34    }
35  }
```

OgreBullet...

El listado anterior muestra además cómo acceder al objeto del mundo de bullet, que permite utilizar gran cantidad de métodos que no están implementados en OgreBullet.

En la línea (2) se obtiene el puntero directamente a la clase *btCollisionWorld*, que se encuentra oculta en la implementación de OgreBullet. Con este puntero se accederá directamente a la funcionalidad de Bullet sin emplear la clase de recubrimieneto de OgreBullet. La clase *btCollisionWorld* sirve a la vez como interfaz y como contenedor de las funcionalidades relativas a la detección de colisiones.

Mediante la llamada a *getDispatcher* (línea (3)) se obtiene un puntero a la clase *btDispather*, que se utiliza en la fase de colisión *broadphase* para la gestión de pares de colisión. Esta clase nos permite obtener el número de *colectores* que hay activos en cada instante. El bucle de las líneas (5-34) se encarga de iterar sobre los colectores. En la línea (6-7) se obtiene un puntero a un objeto de la clase *btPersistentManifold*. Esta clase es una implementación de una caché persistente mientras los objetos colisionen en la etapa de colisión *broadphase*.

Los puntos de contacto se crean en la etapa de detección de colisiones fina (*narrow phase*). La cache de *btPersistentManifold* puede estar vacía o contener hasta un máximo de 4 puntos de colisión. Los algoritmos de detección de la colisión añaden y eliminan puntos de esta caché empleando ciertas heurísticas que limitan el máximo de puntos a 4. Es posible obtener el número de puntos de contacto asociados a la cache en cada instante mediante el método *getNumContacts()*.

La cache de colisión mantiene punteros a los dos objetos que están colisionando. Estos objetos pueden obtenerse mediante la llamada a métodos *get* (líneas (8-11)).

La clase *CollisionsWorld* de *OgreBullet* proporciona un método *findObject* que permite obtener un puntero a objeto genérico a partir de un *SceneNode* o un *btCollisionObject* (ver líneas (15-18)).

La última parte del código (líneas (20-32)) comprueba si alguno de los dos objetos de la colisión son el sumidero. En ese caso, se obtiene el puntero al otro objeto (que se corresponderá con un objeto de tipo esfera creado dinámicamente), y se elimina de la escena. Así, los objetos en esta segunda versión del ejemplo no llegan a añadirse en la caja de la parte inferior del circuito.

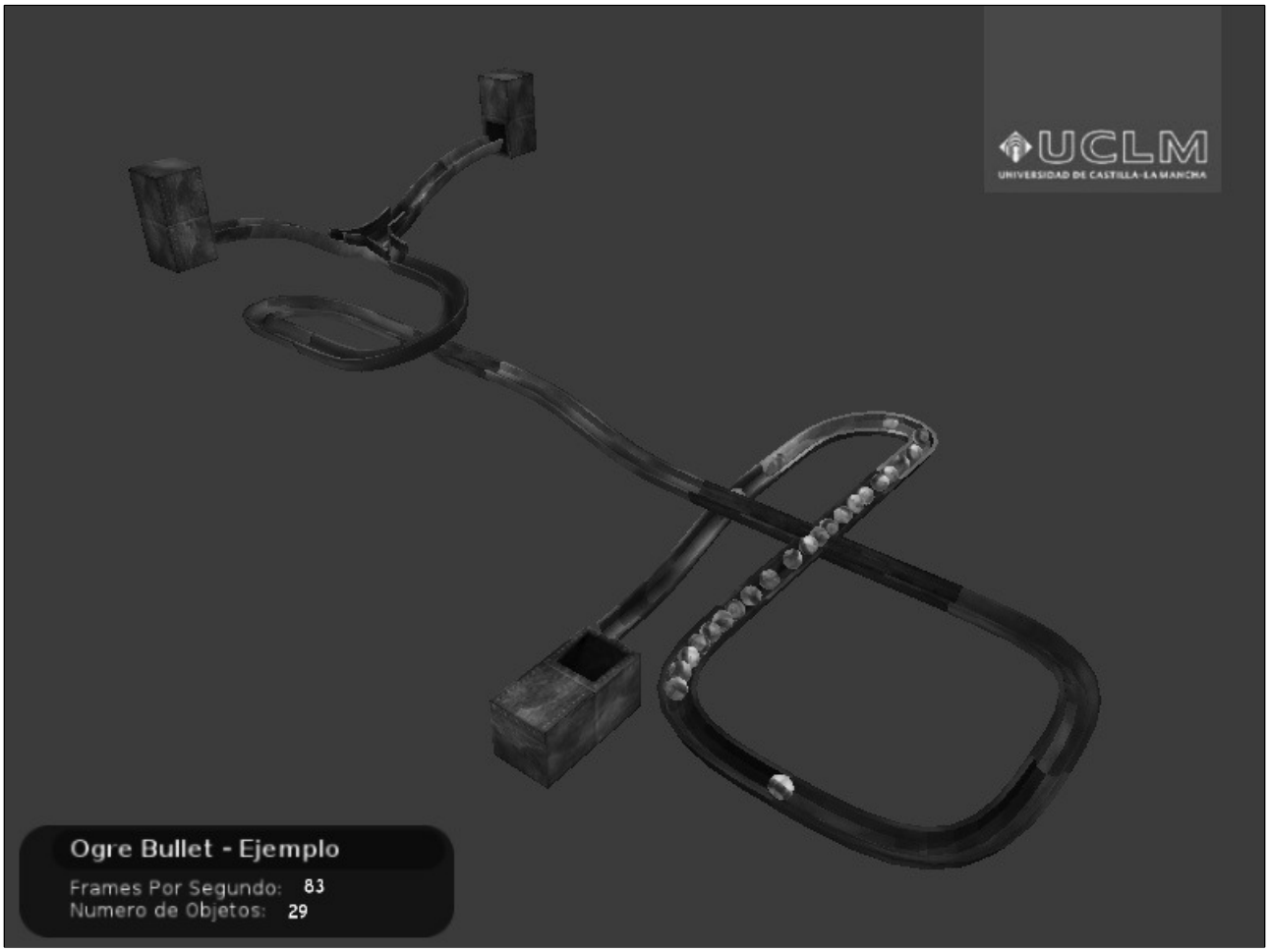


Figura 19.30: Ejemplo de detección de colisiones empleando colectores de contactos.

Otros mecanismos de colisión. En la documentación de Bullet se comentan brevemente otros mecanismos que pueden utilizarse para la detección de colisiones, como los objetos de la clase *btGhostObject*. Los objetos de esta clase pueden tener asociadas llamadas de *callback* de modo que se invoquen automáticamente cuando los objetos se solapen en la etapa de detección de colisiones mediante el test de cajas AABB.

19.11. Restricción de Vehículo

En esta sección estudiaremos cómo utilizar un tipo de restricción específica para la definición de vehículos. OgreBullet cuenta con abstracciones de alto nivel que trabajan internamente con llamadas a las clases derivadas *btRaycastVehicle*, que permiten convertir un cuerpo rígido en un vehículo.

Figura 19.31: Ejemplo de definición de un vehículo en OgreBullet.

A continuación estudiaremos algunos fragmentos de código empleados en el siguiente ejemplo para la construcción del vehículo. Queda como ejercicio propuesto añadir obstáculos y elementos de interacción en la escena empleando mallas triangulares estáticas.

Listado 19.16: Fragmento de MyFrameListener.h

```
1 OgreBulletDynamics::WheeledRigidBody  *mCarChassis;
2 OgreBulletDynamics::VehicleTuning     *mTuning;
3 OgreBulletDynamics::VehicleRayCaster  *mVehicleRayCaster;
4 OgreBulletDynamics::RaycastVehicle    *mVehicle;
5 Ogre::Entity    *mChassis;
6 Ogre::Entity    *mWheels[4];
7 Ogre::SceneNode *mWheelNodes[4];
8 float mSteering;
```

En el anterior archivo de cabecera se definen ciertas variables miembro de la clase que se utilizarán en la definición del vehículo. *mCarChassis* es un puntero a una clase que ofrece OgreBullet para la construcción de vehículos con ruedas. La clase *VehicleTuning* es una clase de cobertura sobre la clase *btVehicleTuning* de Bullet que permite especificar ciertas propiedades del vehículo (como la compresión, suspensión, deslizamiento, etc).

VehicleRayCaster es una clase que ofrece un interfaz entre la simulación del vehículo y el *RayCasting* (usado para localizar el punto de contacto entre el vehículo y el suelo). La clase *RaycastVehicle* es una clase de cobertura sobre la clase base de Bullet *btRaycastVehicle*. Las líneas 5-7 definen los nodos y entidades necesarias para el chasis y las ruedas del vehículo. Finalmente, la varible *Steering* de la línea 8 define la dirección del vehículo.

A continuación estudiaremos la definición del vehículo en el método *CreateInitialWorld* del *FrameListener*. La línea [1] del siguiente listado define el vector de altura del chasis (elevación sobre el suelo), y la altura de conexión de las ruedas en él (línea [2]) que será utilizado más adelante. En la construcción inicial del vehículo se establece la dirección del vehículo como 0.0 (línea [3]).

La líneas [5-9] crean la entidad del chasis y el nodo que la contendrá. La línea [10] utiliza el vector de altura del chasis para posicionar el nodo del chasis.

Listado 19.17: Fragmento de CreateInitialWorld (I).

```
1  const Ogre::Vector3 chassisShift(0, 1.0, 0);
2  float connectionHeight = 0.7f;
3  mSteering = 0.0;
4
5  mChassis = _sceneManager->createEntity("chassis", "chassis.mesh");
6  SceneNode *node = _sceneManager->getRootSceneNode()->
       createChildSceneNode ();
7
8  SceneNode *chassisnode = node->createChildSceneNode();
9  chassisnode->attachObject (mChassis);
10 chassisnode->setPosition (chassisShift);
```

El chasis tendrá asociada una forma de colisión de tipo caja (línea [1]). Esta caja formará parte de una forma de colisión compuesta, que se define en la línea [2], y a la que se añade la caja anterior desplazada según el vector *chassisShift* (línea [3]).

En la línea [4] se define el cuerpo rígido del vehículo, al que se asocia la forma de colisión creada anteriormente (línea [6]). En la línea [9] se establecen los valores de suspensión del vehículo, y se evita que el vehículo pueda desactivarse (línea [8]), de modo que el objeto no se «dormirá» incluso si se detiene durante un tiempo continuado.

Listado 19.18: Fragmento de CreateInitialWorld (II).

```
1 BoxCollisionShape* chassisShape = new BoxCollisionShape(Ogre::
      Vector3(1.f,0.75f,2.1f));
2 CompoundCollisionShape* compound = new CompoundCollisionShape();
3 compound->addChildShape(chassisShape, chassisShift);
4 mCarChassis = new WheeledRigidBody("carChassis", _world);
5 Vector3 CarPosition = Vector3(0, 0, -15);
6 mCarChassis->setShape (node, compound, 0.6, 0.6, 800, CarPosition,
      Quaternion::IDENTITY);
7 mCarChassis->setDamping(0.2, 0.2);
8 mCarChassis->disableDeactivation();
```

En el siguiente fragmento se comienza definiendo algunos parámetros de tuning del vehículo (línea [1]). Estos parámetros son la rigidez, compresión y amortiguación de la suspensión y la fricción de deslizamiento. La línea [5] establece el sistema de coordenadas local del vehículo mediante los índices derecho, superior y adelante.

Las líneas [7] y [8] definen los ejes que se utilizarán como dirección del vehículo y en la definición de las ruedas.

El bucle de las líneas [10-16] construye los nodos de las ruedas, cargando 4 instancias de la malla «*wheel.mesh*».

El ejemplo desarrollado en esta sección trabaja únicamente con las dos ruedas delanteras (índices 0 y 1) como ruedas motrices. Además, ambas ruedas giran de forma simétrica según la variable de dirección *mSteering*. Queda propuesto como ejercicio modificar el código de esta sección para que la dirección se pueda realizar igualmente con las ruedas traseras, así como incorporar otras opciones de motricidad (ver Listado de *FrameStarted*).

Listado 19.19: Fragmento de CreateInitialWorld (III).

```
1  mTuning = new VehicleTuning(20.2, 4.4, 2.3, 500.0, 10.5);
2  mVehicleRayCaster = new VehicleRayCaster(_world);
3  mVehicle = new RaycastVehicle(mCarChassis, mTuning,
       mVehicleRayCaster);
4
5  mVehicle->setCoordinateSystem(0, 1, 2);
6
7  Ogre::Vector3 wheelDirectionCS0(0,-1,0);
8  Ogre::Vector3 wheelAxleCS(-1,0,0);
9
10 for (size_t i = 0; i < 4; i++) {
11   mWheels[i] = _sceneManager->createEntity("wheel"+i,"wheel.mesh");
12   mWheels[i]->setCastShadows(true);
13
14   mWheelNodes[i] = _sceneManager->getRootSceneNode()->
         createChildSceneNode();
15   mWheelNodes[i]->attachObject (mWheels[i]);
16 }
```

El siguiente fragmento de listado se repite para cada rueda, calculando el punto de conexión en función del ancho de cada rueda y la altura de conexión. Este punto es pasado al método *addWheel*, junto con información relativa a ciertas propiedades físicas de cada rueda. La variable *isFrontWheel* (ver línea [3]) indica si la rueda añadida forma parte del conjunto de ruedas delanteras (en este caso, únicamente las dos primeras ruedas tendrán esta variable a *true* en el momento de creación.

Listado 19.20: Fragmento de CreateInitialWorld (IV).

```
1 Ogre::Vector3 connectionPointCS0 (1-(0.3*gWheelWidth),
      connectionHeight, 2-gWheelRadius);
2
3 mVehicle->addWheel(mWheelNodes[0], connectionPointCS0,
      wheelDirectionCS0, wheelAxleCS, gSuspensionRestLength,
      gWheelRadius, isFrontWheel, gWheelFriction, gRollInfluence);
```

Finalmente el método de callback del *FrameStarted* se encarga de modificar la fuerza que se aplica sobre el motor del vehículo cuando se utilizan los cursores superior e inferior del teclado. De igual modo, empleando los cursores izquierdo y derecho del teclado se modifica la dirección del vehículo (ver líneas [11-15]).

Listado 19.21: Fragmento de FrameStarted.

```
1  bool MyFrameListener::frameStarted(const Ogre::FrameEvent& evt) {
2    // Omitido el codigo anterior...
3    mVehicle->applyEngineForce (0,0);
4    mVehicle->applyEngineForce (0,1);
5
6    if (_keyboard->isKeyDown(OIS::KC_UP)) {
7      mVehicle->applyEngineForce (gEngineForce, 0);
```

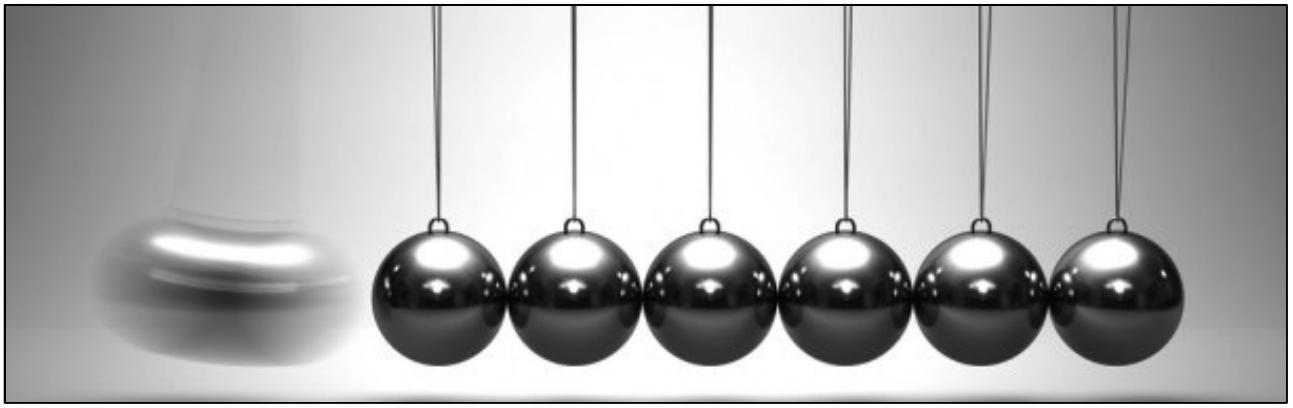


Figura 19.32: La gestión del determinismo puede ser un aspecto crítico en muchos videojuegos. El error de determinismo rápidamente se propaga haciendo que la simulación obtenga diferentes resultados.

```
 8      mVehicle->applyEngineForce (gEngineForce, 1);
 9    }
10
11    if (_keyboard->isKeyDown(OIS::KC_LEFT)) {
12      if (mSteering < 0.8) mSteering+=0.01;
13      mVehicle->setSteeringValue (mSteering, 0);
14      mVehicle->setSteeringValue (mSteering, 1);
15    }
16
17    // Omitido el resto del codigo...
18  }
```

19.12. Determinismo

El determinismo en el ámbito de la simulación física puede definirse de forma intuitiva como la posibilidad de «repetición» de un mismo comportamiento. En el caso de videojuegos esto puede ser interesante en la repetición de una misma jugada en un videojuego deportivo, o en la ejecución de una misma simulación física en los diferentes ordenadores de un videojuego multijugador. Incluso aunque el videojuego siga un enfoque con cálculos de simulación centrados en el servidor, habitualmente es necesario realizar ciertas interpolaciones del lado del cliente para mitigar los efectos de la latencia, por lo que resulta imprescindible tratar con enfoques deterministas.

Para lograr determinismo es necesario lograr que la simulación se realice *exactamente* con los mismos datos de entrada. Debido a la precisión en aritmética en punto flotante, es posible que $v \times 2 \times dt$ no de el mismo resultado que $v \times dt + v \times dt$. Así, es necesario emplear *el mismo* valor de dt en todas las simulaciones. Por otro lado, utilizar un dt fijo hace que no podamos representar la simulación de forma independiente de las capacidades de la máquina o la carga de representación gráfica concreta en cada momento. Así, nos interesa tener lo mejor de ambas aproximaciones; por un lado un tiempo fijo para conseguir el determinismo en la simulación, y por otro lado la gestión con diferentes tiempos asociados al framerate para lograr independencia de la máquina.

Una posible manera de realizar la simulación sería la siguiente: el motor de simulación física se ejecuta por adelantado en intervalos de tiempo discretos dt, de modo que se mantengan los incrementos del motor gráfico con un intervalo adecuado. Por ejemplo, si queremos tener 50fps y la simulación física se ejecuta a 100fps, entonces tendríamos que ejecutar dos veces la simulación física por cada despliegue gráfico.

Esto es correcto con esos cálculos sencillos, pero ¿qué ocurre si queremos dibujar a 200fps?. En ese caso tendríamso que ejecutar la mitad de veces el simulador físico, pero no podemos calcular por adelantado un valos de dt. Además, podría ocurrir que no existiera un múltiplo cómodo para sincronizar el motor de simulación física y el motor de despliegue gráfico.

La forma de resolver el problema pasa por cambiar el modo de pensar en él. Podemos pensar que el motor de render produce tiempo, y el motor de simulación física tiene que consumirlo en bloques discretos de un tamaño determinado.

Puede ayudar pensar que el motor de render *produce* chunks de tiempo discreto, mientras que el motor de simulación física los consume.

A continuación se muestra un sencillo *game loop* que puede emplearse para conseguir determinismo de una forma sencilla.

Los tiempos mostrados en este pseudocódigo se especifican en milisegundos, y se obtienen a partir de una hipotética función *getMillisecons()*.

La línea [1] define *TickMs*, una variable que nos define la velocidad del reloj interno de nuestro juego (por ejemplo, 32ms). Esta variable no tiene que ver con el reloj de Bullet. Las variables relativas al reloj de simulación física describen el comportamiento independiente y asíncrono del reloj interno de Bullet (línea [2]) y el reloj del motor de juego (línea [3]).

Listado 19.22: Pseudocódigo física determinista.

```
1  const unsigned int TickMs 32
2  unsigned long time_physics_prev, time_physics_curr;
3  unsigned long time_gameclock;
4
5  // Inicialmente reseteamos los temporizadores
6  time_physics_prev = time_physics_curr = getMilliseconds();
7  time_gameclock = getMilliseconds();
8
9  while (1) {
10     video->renderOneFrame();
11     time_physics_curr = getMilliseconds();
12     mWorld->stepSimulation(((float)(time_physics_curr -
13                             time_physics_prev))/1000.0, 10);
14     time_physics_prev = time_physics_curr;
15     long long dt = getMilliseconds() - time_gameclock;
16
17     while(dt >= TickMs) {
18         dt -= TickMs;
19         time_gameclock += TickMs;
20         input->do_all_your_input_processing();
21     }
22 }
```

Como se indica en las líneas [6-7], inicialmente se resetean los temporizadores. El pseudocódigo del bucle principal del juego se resume en las líneas [9-21]. Tras representar un frame, se obtiene el tiempo transcurrido desde la última simulación física (línea [11]), y se avanza un paso de simulación en segundos (como la llamada al sistema lo obtiene en milisegundos y Bullet lo requiere en segundos, hay que dividir por 1000).

Por último, se actualiza la parte relativa al reloj de juego. Se calcula en dt la diferencia entre los milisegundos que pasaron desde la última vez que se acualizó el reloj de juego, y se dejan pasar (en el bucle definido en las líneas [17-21]) empleando ticks discretos. En cada tick consumido se procesan los eventos de entrada.

19.13. Escala de los Objetos

Como se ha comentado en secciones anteriores, Bullet asume que las unidades de espacio se definen en metros y el tiempo en segundos. El movimieneto de los objetos se define entre 0.05 y 10 unidades. Así, la escala habitual para definir los pasos de simulación suelen ser 1/60 segundos. Si los objetos son muy grandes, y se trabaja con la gravedad por defecto ($9{,}8m/s^2$), los objetos parecerán que se mueven a cámara lenta. Si esto ocurre, muy probablemente tengamos un problema relativo a la escala de los mismos.

Una posible solución puede pasar por aplicar una escala al mundo de la simulación. Esto esquivale a utilizar un conjunto diferente de unidades, como centímetros en lugar de metros. Si se seleccionan con cuidado, esto puede permitir realizar simulaciones más realistas. Por ejemplo, si queremos diseñar un videojuego de billar, escalamos el mundo en un factor de 100, de modo que 4 unidades equivaldrán a 4cm (diámetro de las bolas de billar).

19.14. Serialización

La serialización de objetos en Bullet es una característica propia de la biblioteca que no requiere de ningún plugin o soporte adicional. La serialización de objetos presenta grandes ventajas relativas al precálculo de formas de colisión complejas. Para guardar un mundo dinámico en un archivo .bullet, puede utilizarse el siguiente fragmento de código de ejemplo:

Listado 19.23: Ejemplo de Serialización.

```
1 btDefaultSerializer*    serializer = new btDefaultSerializer();
2 dynamicsWorld->serialize(serializer);
3
4 FILE* file = fopen("testFile.bullet","wb");
5 fwrite(serializer->getBufferPointer(),serializer->
      getCurrentBufferSize(),1, file);
6 fclose(file);
```

Aunque lo más sencillo es serializar un mundo completo, es igualmente posible serializar únicamente algunas partes del mismo. El foramto de los archivos *.bullet* soporta la serialización parcial de elementos empleando chunks independientes.

En la posterior carga de los archivos .bullet, se debe utilizar la cabecera de *BulletWorldImporter*, creando un objeto de esa clase. El constructor de esa clase requiere que se le especifique el mundo dinámico sobre el que creará los objetos que fueron serializados. El uso del importador puede resumirse en el siguiente fragmento de código:

Listado 19.24: Importación de datos serializados.

```
1 #include "btBulletWorldImporter.h"
2 btBulletWorldImporter* f = new btBulletWorldImporter(_dynWorld);
3 f->loadFile("testFile.bullet");
```

Módulo #3

Técnicas Avanzadas

Técnicas Avanzadas

El objetivo de este módulo, titulado «*Técnicas Avanzadas*» dentro del Curso de Experto en Desarrollo de Videojuegos, es profundizar es aspectos de desarrollo más avanzados que complementen el resto de contenidos de dicho curso y permitan explorar soluciones más eficientes en el contexto del desarrollo de videojuegos. En este módulo se introducen aspectos básicos de jugabilidad y se describen algunas metodologías de desarrollo de videojuegos. Así mismo, también se estudian los fundamentos básicos de la validación y pruebas en este proceso de desarrollo. No obstante, uno de los componentes más importantes del presente módulo está relacionado con aspectos avanzados del lenguaje de programación C++, como por ejemplo el estudio en profundidad de la biblioteca STL, y las optimizaciones. Finalmente, el presente módulo se complementa con aspectos de representación avanzada, como los filtros de partículas o la programación de shaders, y con un estudio en detalle de técnicas de optimización para escenarios interiores y exteriores. Por otra parte, se realiza un estudio de la plataforma de desarrollo de videojuegos *Unity*, especialmente ideada para el desarrollo de juegos en plataformas móviles.

Capítulo 20

Aspectos de Jugabilidad y Metodologías de Desarrollo

Miguel Ángel Redondo Duque
José Luis González

En este capítulo se introducen aspectos básicos relativos al concepto de jugabilidad, como por ejemplo aquellos vinculados a su caractarización en el ámbito del desarrollo de videojuegos o las facetas más relevantes de los mismos, haciendo especial hincapié en la parte relativa a su calidad.

Por otra parte, en este capítulo también se discuten los fundamentos de las metodologías de desarrollo para videojuegos, estableciendo las principales fases y las actividades desarrolladas en ellas.

20.1. Jugabilidad y Experiencia del Jugador

20.1.1. Introducción

En el desarrollo de sistemas interactivos es fundamental la participación del usuario. Por ello, se plantean los denominados métodos de *Diseño Centrado en el Usuario* que se aplican, al menos, al desarrollo software que soporta directamente la interacción con el usuario. En otras palabras, es fundamental contar con su participación para que tengamos la garantía de que se le va a proporcionar buenas experiencias de uso. El software para videojuegos se puede considerar como un caso particular de sistema interactivo por lo que requiere de un planteamiento similar en este sentido, aunque en este ámbito, los términos y conceptos que se emplean para este propósito, varían ligeramente.

En el desarrollo de videojuegos es importante tener siempre presente que hay que lograr que el jugador sienta las mejores experiencias (entretenimiento, diversión, et.) posibles durante su utilización . El incremento de estas experiencias revierte directamente en el éxito del videojuego. Así pues, es conveniente conocer las propiedades que caracterizan dichas experiencias, poder medirlas durante el proceso de desarrollo y así asegurar su éxito y calidad. En adelante, nos referiremos a esto como *Experiencia del Jugador*.

La Experiencia del Jugador suele medirse utilizándose el concepto de *Jugabilidad* como propiedad característica de un videojuego, aunque su caracterización y forma de medirla no es algo plenamente formalizado e implantado en la industria del desarrollo de videojuegos.

Como paso previo para entender los conceptos de Jugabilidad y, por extensión, de Experiencia del Jugador, conviene repasar un concepto fundamental en el ámbito de los sistemas interactivos que es la Usabilidad. La Usabilidad se refiere a *la capacidad de un software de ser comprendido, aprendido, usado y ser satisfactorio para el usuario, en condiciones especificas de uso o la eficiencia y satisfacción con la que un producto permite alcanzar objetivos específicos a usuarios específicos en un contexto de uso especifico*. Para entender y medir la Usabildiad, se han identificado una serie de propiedades como son: efectividad, eficiencia, satisfacción, aprendizaje y seguridad [4] [85] [66] [28]. Estas son las propiedades que son objeto de medición y, a partir de ellas, se puede valorar el grado de Usabilidad de un sistema.

El desarrollo de software usable redunda directamente en reducción de costes de producción, optimización del mantenimiento e incremento de la calidad final del producto. Además, las propiedades que caracterizan la Usabilidad influyen muy directamente en el uso que los usuarios hacen, contribuyendo incrementar su satisfacción, su productividad en la realización de tareas y reduciendo su nivel de estrés. En definitiva, la Usabilidad puede considerarse como un reflejo de la Experiencia del Usuario en un sistema interactivo que soporta la realización de una serie de tareas específicas para lograr un objetivo bien definido.

Según Nielsen Norman Group se define la *Experiencia del Usuario como la sensación, sentimiento, respuesta emocional, valoración y satisfacción del usuario respecto a un producto, resultado del proceso de interacción con el producto y de la interacción con su proveedor* [67]. En este sentido, cabe destacar la importancia que juegan diversos conceptos como la utilidad, la usabilidad, la deseabilidad, la accesibilidad, facilidad de uso, lo valioso del producto y lo creíble que pueda ser para el usuario. La Experiencia de Usuario está estrechamente relacionada con el contexto de uso del sistema interactivo, el contenido manipulado y los usuarios que lo usan. Lo que significa que variando alguno de estos elementos, el resultado puede ser totalmente diferente e incluso opuesto.

La relación que existe entre Experiencia de Usuario y Usabilidad puede considerarse equivalente a la que existe entre Experiencia del Jugador y Jugabilidad, aunque no se trata de una simple traslación del dominio de aplicación. Así lo vamos a considerar para explicar cómo se puede caracterizar la Jugabilidad y que en base a su medición se obtenga una valoración de la Experiencia del Jugador. Además, se apuntarán algunas ideas metodológicas orientadas a lograr mejores desarrollos de videojuegos, desde el punto de vista de la Jugabilidad.

20.1.2. Caracterización de la Jugabilidad

La Jugabilidad extiende el concepto de Usabilidad, pero no se reduce únicamente la idea de Usabilidad en el caso particular de los videojuegos. Tampoco sería correcto reducirla únicamente al grado de diversión de un juego. Para diferenciar claramente este concepto que es un tanto difuso, lo adecuado es representarlo por un conjunto de

atributos o propiedades que lo caracterizan. Estos atributos podrán ser medidos y valorados, para así comparar y extraer conclusiones objetivas. Este trabajo fue realizado por José Luis González [40] que define la Jugabilidad como *el conjunto de propiedades que describen la experiencia del jugador ante un sistema de juego determinado, cuyo principal objetivo es divertir y entretener "de forma satisfactoria y creíble", ya sea solo o en compañía*.

Es importante remarcar los conceptos de satisfacción y credibilidad. El primero es común a cualquier sistema interactivo. Sin embargo, la credibilidad dependerá del grado en el que se pueda lograr que los jugadores se impliquen en el juego.

Hay que significar que los atributos y propiedades que se utilizan para caracterizar la Jugabilidad y la Experiencia del Jugador, en muchos casos ya se han utilizado para caracterizar la Usabilidad, pero en los videojuegos presentan matices distintos. Por ejemplo, el "Aprendizaje" en un videojuego puede ser elevado, lo que puede provocar que el jugador se vea satisfecho ante el reto que supone aprender a jugarlo y, posteriormente, desarrollar lo aprendido dentro del juego. Un ejemplo lo tenemos en el videojuego *Prince of Persia*, donde es difícil aprender a controlar nuestro personaje a través de un mundo virtual, lo que supone un reto en los primeros compases del juego. Sin embargo, en cualquier otro sistema interactivo podría suponer motivo suficiente de rechazo. Por otro lado, la "Efectividad" en un juego no busca la rapidez por completar una tarea, pues entra dentro de la naturaleza del videojuego que el usuario esté jugando el máximo tiempo posible y son muchos los ejemplos que podríamos citar.

Los atributos a los que hacemos referencia para caracterizar la Jugabilidad son los siguientes:

- **Satisfacción**. Agrado o complacencia del jugador ante el videojuego y el proceso de jugarlo.
- **Aprendizaje**. Facilidad para comprender y dominar el sistema y la mecánica del videojuego. Más adelante se indica cómo estos conceptos se definen en lo que se denomina Gameplay y que se construye durante el proceso de desarrollo del juego.
- **Efectividad**. Tiempo y recursos necesarios para ofrecer diversión al jugador mientras éste logra los objetivos propuestos en el videojuego y alcanza su meta final.
- **Inmersión**. Capacidad para creerse lo que se juega e integrarse en el mundo virtual mostrado en el juego.
- **Motivación**. Característica del videojuego que mueve a la persona a realizar determinadas acciones y a persistir en ellas para su culminación.
- **Emoción**. Impulso involuntario originado como respuesta a los estímulos del videojuego, que induce sentimientos y que desencadena conductas de reacción automática.
- **Socialización**. Atributos que hacen apreciar el videojuego de distinta manera al jugarlo en compañía (multijugador), ya sea de manera competitiva, colaborativa o cooperativa.

La figura 20.1 muestra como estos atributos y algunos otros más pueden estar relacionados con el concepto de Usabilidad tal y como se recoge en las normas ISO/IEC-9241. Hay algunos atributos que están relacionados con el videojuego (producto) y otros se vinculan al proceso de desarrollo del juego (desarrollo), algunos hacen referencia a su influencia sobre el jugador/es (usuarios o grupos de usuarios).

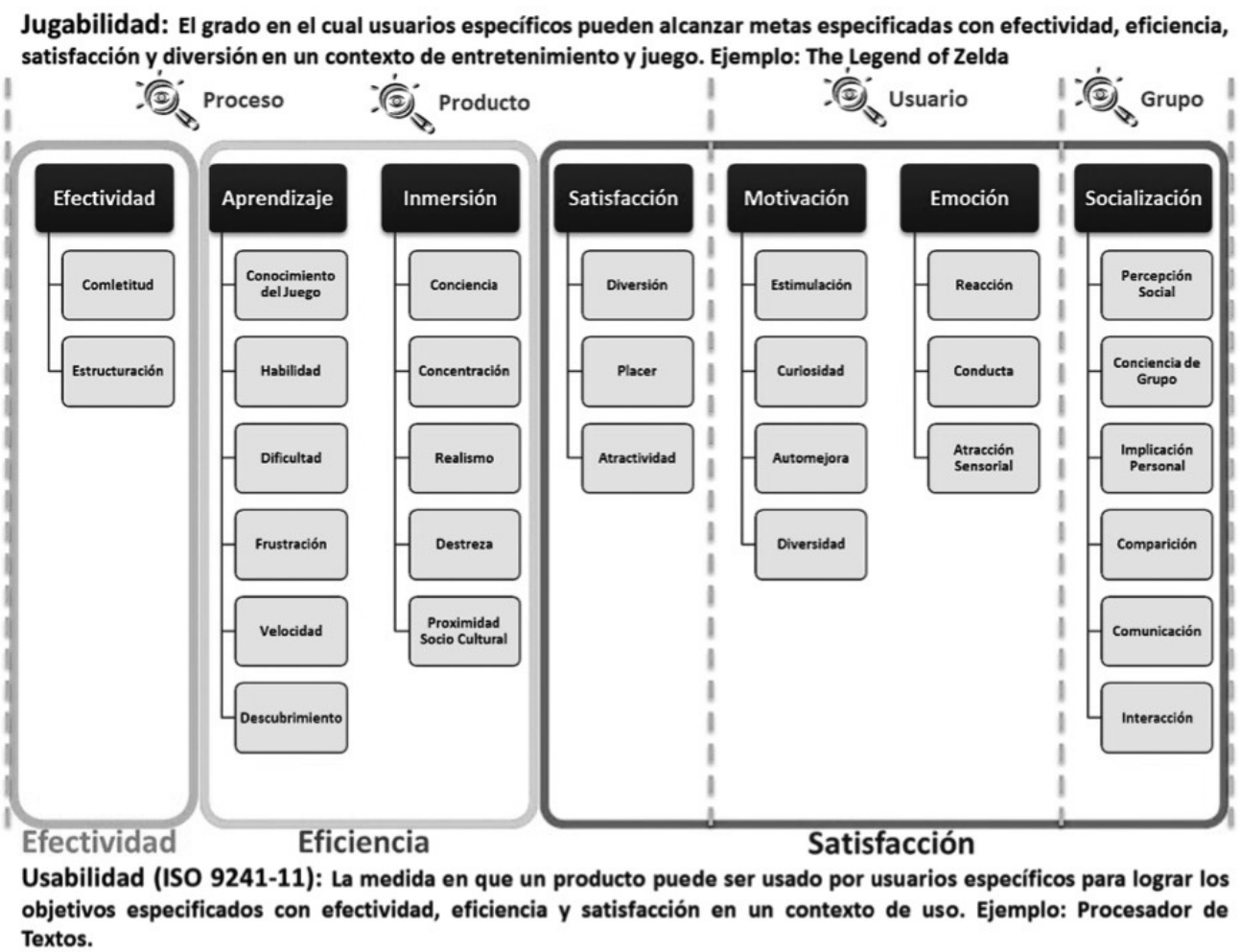


Figura 20.1: Relación entre atributos de Usabilidad y de Jugabilidad

20.1.3. Facetas de la Jugabilidad

Uno de los objetivos, una vez definida la Jugabilidad, es poder medirla o cuantificarla. Este proceso es costoso debido a la cantidad de objetivos no funcionales que afectan a la Experiencia del Jugador. Como plantea [40], una buena estrategia es la de considerar una representación de la Jugabilidad basada en facetas de la misma. La organización en facetas puede considerarse una subdivisión lógica de la Jugabilidad global en jugabilidades un poco más específicas. Cada una de estas facetas facilitará la identificación y medición de las propiedades introducidas anteriormente. Además, así será más fácil relacionar la Jugabilidad con los elementos particulares de un videojuego.

Como facetas particulares podrían considerarse las siguientes, aunque no es algo cerrado y en algún juego particular podría aparecer y proponerse alguna otra faceta que fuese objeto de consideración:

- **Jugabilidad Intrínseca**. Se trata de la Jugabilidad medida en la propia naturaleza del juego y cómo se proyecta al jugador. Está ligada al diseño del *Gameplay* que se describe más adelante. La forma de valorarla pasa por analizar cómo se representan las reglas, los objetivos, el ritmo y las mecánicas del videojuego.
- **Jugabilidad Mecánica**. Es la Jugabilidad asociada a la calidad del videojuego como sistema software. Está ligada a lo que sería el motor del juego, haciendo hincapié en características como la fluidez de las escenas cinemáticas, la correcta iluminación, el sonido, los movimientos gráficos y el comportamiento de los personajes del juego y del entorno, sin olvidar los sistemas de comunicación en videojuegos multijugador.
- **Jugabilidad Interactiva**. Es la faceta asociada a todo lo relacionado con la interacción con el usuario, el diseño de la interfaz de usuario, los mecanismos de diálogo y los sistemas de control.
- **Jugabilidad Artística**. Está asociada a la calidad y adecuación artística y estética de todos los elementos del videojuego y a la naturaleza de éste. Entre ellos estarán la calidad gráfica y visual, los efectos sonoros, la banda sonora y las melodías del juego, la historia y la forma de narración de ésta, así como la ambientación realizada de todos estos elementos dentro del videojuego.
- **Jugabilidad Intrapersonal** (o Personal). Está relacionada con la percepción que tiene el propio usuario del videojuego y los sentimientos que a éste le produce. Como tal, tiene un alto valor subjetivo.
- **Jugabilidad Interpersonal** (o de Grupo). Muestra las sensaciones o percepciones de los usuarios que aparecen cuando se juega en grupo, ya sea de forma competitiva, cooperativa o colaborativa. En relación a cualquier sistema interactivo con soporte para grupos, se relacionaría con lo que tiene que ver con percepción del grupo (o awareness de grupo).

En [40] incluso se relacionan, a nivel interactivo, estas facetas para ilustrar cómo pueden ser las implicaciones e influencias que presentan. Esta relación se resume en la figura 20.2.

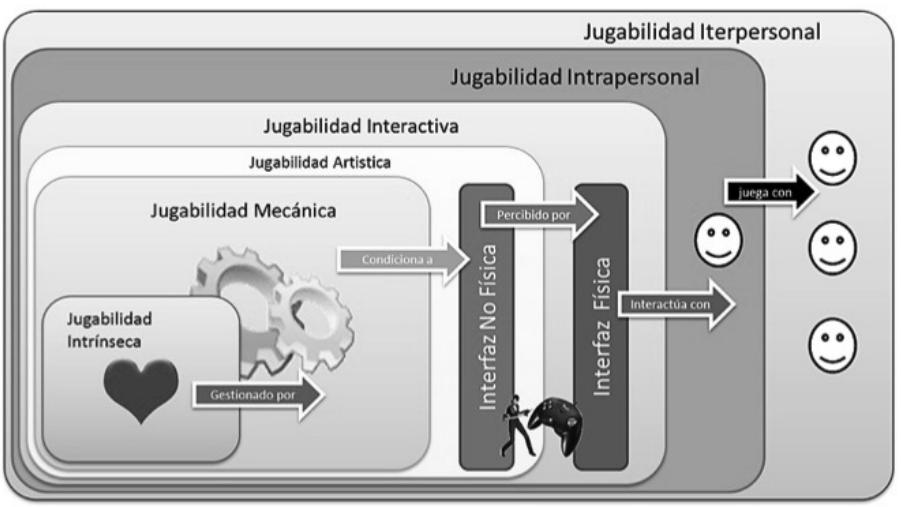


Figura 20.2: Relaciones entre las Facetas de la Jugabilidad

Con todo lo anterior, se puede concluir que la Jugabilidad de un juego podría considerarse como el análisis del valor de cada una de las propiedades y de los atributos en las facetas consideradas.

20.1.4. Calidad de un juego en base a la Jugabilidad

Como ha quedado patente, el análisis de la calidad de un videojuego únicamente a partir de la Usabilidad o de la calidad de uso es insuficiente. Por esta razón, la caracterización de la Experiencia del Jugador en base a la Jugabilidad mediante una serie de propiedades, atributos y facetas proporciona un instrumento adicional. Con esto se pueden obtener medidas de la calidad de las experiencias durante el juego e incluso pueden utilizarse para extender el estándar de calidad ISO 25010:2011 al contexto de los videojuegos.

Se puede destacar que hay una serie de propiedades de la Jugabilidad que influyen directamente en la Calidad del Producto y otras en la Calidad del Proceso de Uso y que, fundamentalmente tienen que ver con la habilidad del jugador para utilizarlo.

La Jugabilidad puede entenderse como la calidad de uso de un videojuego, pero la definición de ciertos atributos de la calidad en uso, según ISO, debe reescribirse adaptándose al contexto de ocio en el que estamos envueltos. Partiendo de estas consideraciones y entrando en mayor detalle respecto de la definición previa, la Jugabilidad representa *el grado por el que usuarios específicos (jugadores) alcanzan metas de un juego con efectividad, eficiencia, flexibilidad, seguridad y, especialmente, satisfacción en un contexto jugable de uso*.

Estas ideas serán las bases para la extensión del modelo de calidad 25010 basándose en el modelo de la Jugabilidad. Definiremos el modelo de calidad en base a los pilares básicos necesarios para ello: propiedades o factores de calidad, métricas y herramientas de evaluación.

Como propiedades o factores de calidad son consideradas las siguientes, siempre y en todos los casos ajustado al contexto de uso concreto que aporta el videojuego objeto de estudio:

- **Efectividad**. La definimos como el grado en el que usuarios específicos (jugadores) pueden lograr las metas propuestas con precisión y completitud en un contexto de uso concreto.
- **Eficiencia**. Es el grado con el que usuarios específicos (jugadores) pueden lograr las metas propuestas invirtiendo una cantidad apropiada de recursos en relación a la efectividad lograda en un contexto de uso concreto. Este factor está determinado por la facilidad de aprendizaje y la inmersión.
- **Flexibilidad**. Es el grado con el que el videojuego se puede usar en distintos contextos posibles o por los distintos perfiles de jugadores y de juego existentes.
- **Seguridad/Prevención**. Nivel aceptable de riesgo para la salud del jugador, o los datos de éste, en un contexto de uso concreto.
- **Satisfacción**. Grado con el que los usuarios (jugadores) están satisfechos en un contexto de uso concreto, el que le aporta un videojuego. En este factor consideramos distintos atributos como: agrado, atracción, placentero, confortable, confiable, motivador, emocionable y sociable.

El diagrama de la figura 20.4 muestra la relación de estas propiedades con los principales conceptos que las definen y caracterizan.

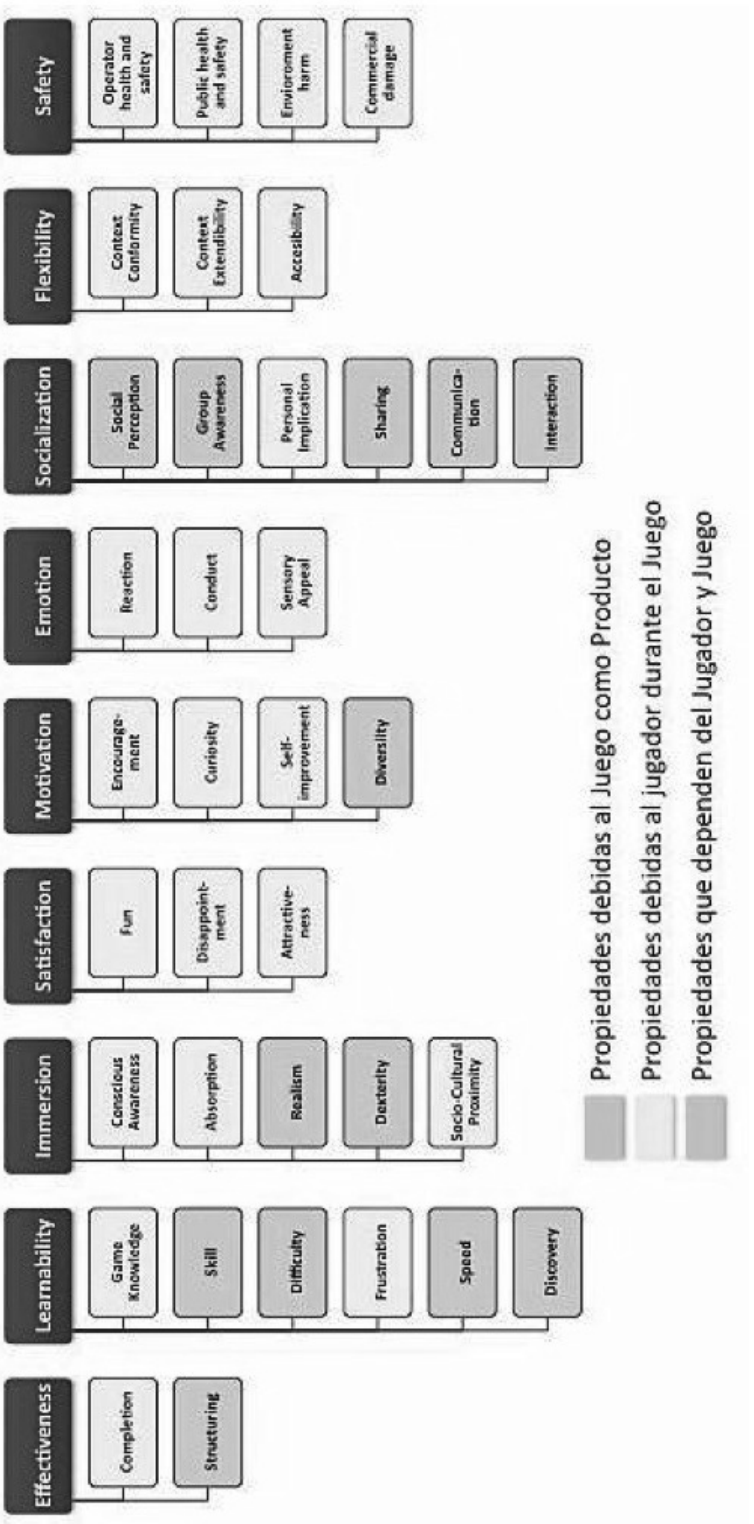


Figura 20.3: Clasificación de las propiedades de la calidad del producto y del proceso en un videojuego

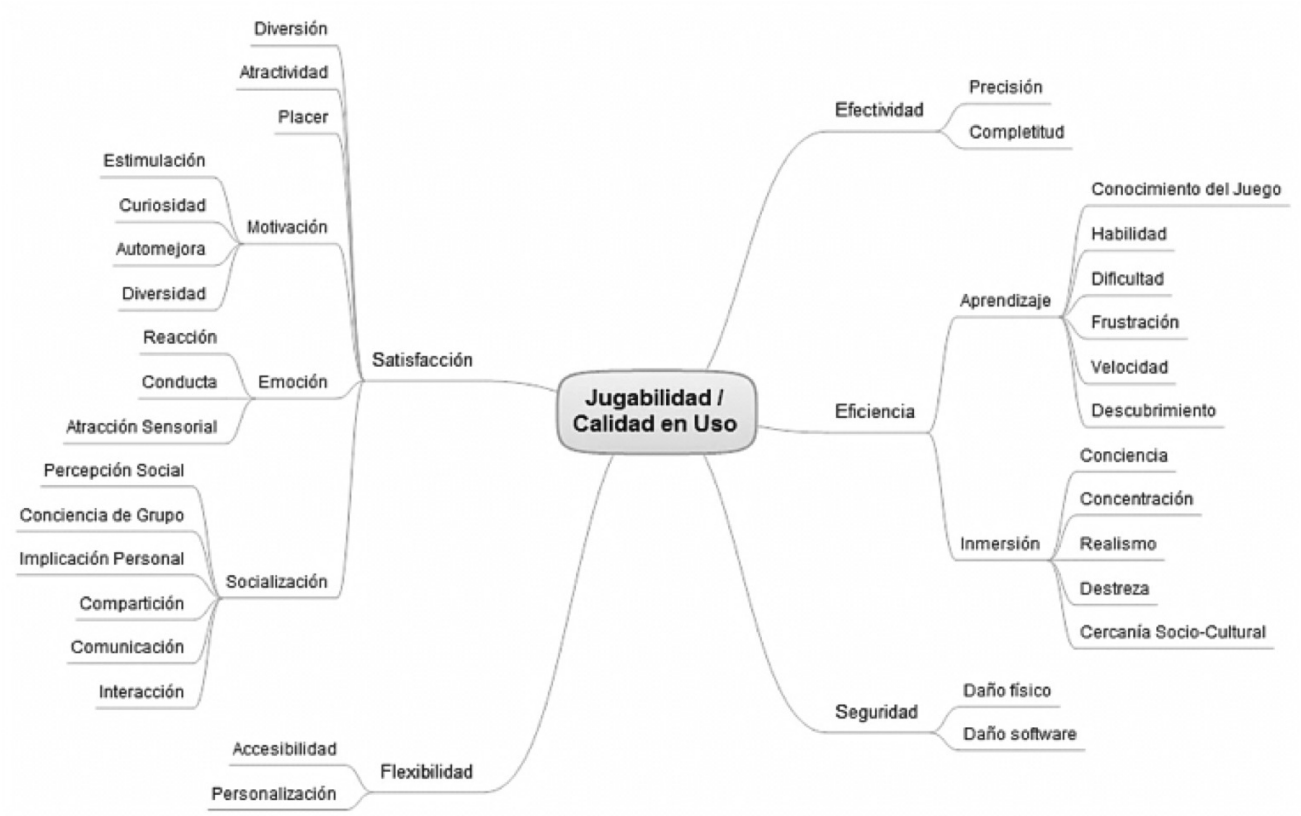


Figura 20.4: Propiedades o factores de calidad y conceptos que las caracterizan

El modelo de Jugabilidad presentado se completa con la identificación y asociación de métricas para los los factores y atributos que hemos apuntado. Estas métricas (ver figuras 20.5 a 20.9) son consecuencia de la adaptación de métricas propuestas en otros estándares internacionales pero particularizadas para el caso particular de los videojuegos.

	Nombre de la Métrica	Propósito	Fórmula	Interpretación	Método de Evaluación
Efectividad	Efectividad en la Meta	¿Qué porcentaje de metas y retos se han alcanzado correctamente?	M1 = |1-ΣAi| Ai Valor proporcional de cada acción incorrecta	M1 ∈ [0, 1], cercano 1, lo mejor	Test de Usuarios
	Completitud de la Meta	¿Qué porcentaje de metas y retos se han completado?	X = A/B A = n. de metas completadas B = n. total de metas intentadas	M1 ∈ [0, 1], cercano 1, lo mejor	Test de Usuarios
	Frecuencia de Intentos por Meta	¿Cuál ha sido la frecuencia de intentos?	X = A/T A = n. de intentos realizados por jugador T = tiempo o número de metas	Jugador experto cercano a 0. Al comienzo > 0	Test de Usuarios

Figura 20.5: Métricas para atributos de Efectividad

	Nombre de la Métrica	Propósito	Fórmula	Interpretación	Método de Evaluación
Eficiencia	Tiempo de Meta	¿Cuánto tiempo requiere el jugador para lograr una meta?	X = Ta	Jugadores novatos necesitan más tiempo	Test de Usuarios
	Eficiencia de Meta	¿Cómo de eficiente es el usuario?	X = M1/T	X ∈ [0, 1], cercano a valores intermedios	Test de Usuarios
	Eficiencia Relativa al Nivel del Usuario	¿Cómo de eficiente es un jugador experto frente a un jugador nuevo?	X = A/B A = eficiencia del jugador normal B = eficiencia del jugador experto	M1 ∈ [0, 1], cercano 1, lo mejor	Test de Usuarios

Figura 20.6: Métricas para atributos de Eficiencia

	Nombre de la Métrica	Propósito	Fórmula	Interpretación	Método de Evaluación
Flexibilidad	Accesibilidad	¿Qué porcentaje de metas se logran utilizando distintas formas de interacción diferentes a las usadas por defecto?	X = A/B A = metas con diferentes métodos de interacción B = n. total de metas	M1 ∈ [0, 1], cercano 1 lo mejor	Test de Usuarios
	Personalización	¿Qué proporción de la personalización disponible utiliza el jugador?	X = A/B A = elementos personalizables B = elementos en el juego	M1 ∈ [0, 1], si cercano a 1 métodos de interacción originales quizás deban ser cambiados	Test de Usuarios

Figura 20.7: Métricas para atributos de Flexibilidad

	Nombre de la Métrica	Propósito	Fórmula	Interpretación	Método de Evaluación
Seguridad	Seguridad y Salud del Jugador	¿Cómo incide en la salud del jugador el uso del producto?	X = 1 – A / B A = n. de jugadores que informan de problemas relacionados con la seguridad B = número total de jugadores	M1 ∈ [0, 1], cercano 1, lo mejor	Test de Usuarios
	Daño software	¿Cómo incide la corrupción del software en el juego?	X = 1 – A / B A = número de veces que el videojuego falla y es detectado por el jugador. B = n. total de situaciones de uso	M1 ∈ [0, 1], cercano 1, lo mejor	Test de Usuarios

Figura 20.8: Métricas para atributos de Seguridad

Las métricas para la *efectividad* están basadas en "metas" ya que en su objetivo principal como mecanismo de entretenimiento, el usuario debe superar unos retos para alcanzar una meta con libertad de acciones dentro de las mecánicas del juego. Por tanto, podríamos decir que lo importante es superar el reto, el cómo dependerá de las

habilidades y maneras de jugar del jugador. Por otro lado, en un videojuego hay metas que debe realizar el jugador, pero sin embargo, la facilidad de consecución de esas metas no es el principal objetivo. De hecho, más bien podría decirse que es justo lo contrario, el uso del juego debe ser una motivación y presentar cierta dificultad de consecución, de lo contrario el jugador perderá motivación por el uso del videojuego.

Así mismo, la medida de la *frecuencia de error* en el software tradicional con un valor cercano a 0 siempre es mejor, pero en videojuegos podemos encontrar tanto valores cercanos a 0 como a 1. Si el valor es cercano a 0, nos encontramos ante un jugador experto o que la dificultad del juego es muy baja. Cercano a 1, nos informa que nos encontramos ante un jugador novato, o que se encuentra en los primeros compases del juego, o que la dificultad es muy elevada. Es por ello que los videojuegos ofrecen distintos niveles de dificultad para atraer a los nuevos jugadores, evitando, por ejemplo, que una dificultad extremadamente fácil haga que el juego pierda interés y se vuelva aburrido.

La *eficacia* en el caso de los videojuegos es relativa, es decir, el usuario querrá jugar de forma inmediata, sin perder tiempo en recibir excesiva información, pero, de la misma manera que comentábamos con anterioridad, el juego debe aportar dificultad y el usuario debería encontrar cierta resistencia y progresiva dificultad en la consecución de las metas que lleve asociado.

La *personalización* también es algo especialmente deseable en el mundo del videojuego, porque en él coexisten muchos elementos de diseño que tratan de distraer, de acompañar y de establecer la forma de interacción. Ésta última debería ser flexible en cuanto a poder dar soporte a diferentes formas de interactuar: teclas, mandos, sonidos, etc. El atributo de la *accesibilidad*, aunque deseable y exigible, tradicionalmente no ha contado con mucha atención en el desarrollo de videojuegos.

Este aspecto está cambiando y la presencia de este atributo contribuye al uso del mismo ya sea en la interfaz de usuario o en las mecánicas del juego. En este modelo de Jugabilidad este atributo se consideró implícitamente dentro de otros. Los problemas de la accesibilidad pueden considerarse problemas de Usabilidad/Jugabilidad para, por ejemplo, jugadores con algún tipo de discapacidad.

Si un jugador no puede entender lo que se dice en determinadas escenas u oír si otro personaje camina detrás de él por problemas de sonido, es recomendable el uso de subtítulos. Si el jugador no puede manejar determinado control de juego, se recomienda el uso de dispositivos alternativos para facilitar el control de juego.

La *seguridad/prevención* es un factor que, en el caso de los videojuegos, toma cada vez más importancia. El juego, en la actualidad, no es sólo estático, mental y de sobremesa, sino que supone, en algunos casos, exigencias físicas, por ejemplo el uso de un control de juego que demande un esfuerzo corporal o movimientos bruscos, los cuales pueden ser potencialmente peligrosos o dañinos para la salud si el jugador desarrolla la actividad de juego con ellos durante un tiempo prolongado de ocio.

La *satisfacción* es el atributo más determinante al tratar con videojuegos. Muchos aspectos: cognitivos, emocionales, físicos, de confianza y sociales pueden considerarse bajo este factor de la Jugabilidad. La estimación de la misma se realiza fundamentalmente con cuestionarios y observando al jugador mientras juega y viendo cuáles son sus preferencias de un momento de ocio para el siguiente. Probablemente, este atributo en videojuegos es el más rico y subjetivo. Por lo tanto, es el que se ha enriquecido más con atributos y propiedades para mejorar su medida y estimación.

Finalmente, en la última columna de la tabla se proponen diversos métodos de evaluación para cada métrica. Estos métodos pueden enriquecerse y guiarse por las facetas, las cuales nos pueden ayudar a identificar la calidad de elementos concretos de un videojuego según el uso y las acciones mostradas por el conjunto de jugadores.

	Nombre de la Métrica	Propósito	Fórmula	Interpretación	Método de Evaluación
Satisfacción	Escala de Satisfacción	¿Cómo de satisfecho está el jugador?	X = A/B A = cuestionario con escala psicométrica B = media popular	X>0 el mayor, lo mejor	Test de Usuarios + Cuestionarios
	Cuestionario de Satisfacción	¿Cómo de satisfecho está el jugador con las características propias del videojuego?	X = ΣAi /n A i= respuesta a la pregunta B = número de respuestas	Comparar con valores previos, o con la media popular	Test de Usuarios + Cuestionarios
	Preferencia de Uso	¿Qué porcentaje de usuarios prefieren el videojuego frente a otro?	X = A/B A = n. de veces que características propias del juego es usada B = n. de veces que jugadores intentan jugar a un juego	M1 ∈ [0, 1], cercano 1, lo mejor	Test de Usuarios + Cuestionarios
	Socialización	¿Qué porcentaje de los retos son resueltos jugando en grupo?	X = A/B A = n. de veces que el juego se usa en un contexto social B = n. de veces que el juego es usado	M1 ∈ [0, 1], cercano 1, juego social, cercano a 0, juego individual	Test de Usuarios + Cuestionarios

Figura 20.9: Métricas para atributos de Satisfacción

Las principales formas de medición son la observación, donde podemos medir con herramientas cómo y de qué manera actúa el jugador con el videojuego, usando por ejemplo las métricas presentadas o usar cuestionarios o tests heurísticos para preguntar o interrogar por atributos de la Jugabilidad. Estos cuestionarios pueden ir guiados por facetas para facilitar su análisis.

20.2. Metodologías de Producción y Desarrollo

Como en el desarrollo de cualquier producto software, para el construcción de un videojuego se requiere tener presente los principios fundamentales de la Ingeniería del Software y, especialmente, la metodología de desarrollo adecuada para el producto que se pretende construir y el contexto en el que se llevará a cabo. Sin embargo, el diseño y desarrollo de un videojuego no sólo se reduce al desarrollo técnico de un producto software sino que supone una actividad multidisciplinar que abarca desde la idea y concepción inicial hasta su versión final. Además, hay que tener presente que el desarrollo de un videojuego suele ser un proyecto de gran envergadura en tiempo y en dinero. Por ejemplo, la producción de *Half-Life 2* supuso más de cuatro años de trabajo y un presupuesto final que se situó alrededor de los cincuenta millones de dólares. En estas situaciones, hay aspectos clave que requieren de una minuciosa planificación y metodología, ya que desde que se concibe un proyecto hasta que se comercializa transcurren grandes periodos de tiempo lo que en el ámbito tecnológico puede ser la causa de presentar importantes desfases y, por lo tanto, desembocar en un estrepitoso fracaso.

Así pues, se puede asegurar que la realización de un videojuego es una tarea delicada que requiere de una metodología específica. Sin embargo, las metodologías claramente establecidas para desarrollo de software no se adaptan a este proceso con garantías de calidad suficientes y no existe en este ámbito un claro planteamiento de cómo afrontar el trabajo. No obstante, son muchos expertos los que coinciden en que el ciclo de vida del desarrollo de videojuegos se debe aproximar al del desarrollo de una película de cine, estableciendo tres fases claramente diferencias que son **Pre-Producción**, **Producción** y **Post-Producción**. A su vez en cada una de estas fases se identifican diversas etapas significativas y el equipo de producción se distribuye para colaborar en cada una de ellas.

El equipo de personas que suelen trabajan en un proyecto de desarrollo de un videojuego comercial de tamaño medio-alto oscina entre 40 y 140. Además, el tiempo que dura el proceso puede llegar a superar los tres años. Teniendo presente esto y, especialmente, su similitud con la producción de una película en [12] se propone una organización de referencia para el equipo de producción. Esta organización es la que aparece en la figura 20.10 y que ha sido traducida en [40].

La organización de las etapas del proceso de producción y la relación entre las mismas da lugar a un modelo de proceso que se asemeja al denominado Modelo en Cascada de Royce [78] en el que se establece la realización secuencial de una serie de etapas, impidiendo el comienzo de una nueva etapa sin la finalización de la anterior. Esta característica sacrifica de forma importante la posibilidad de paralelismo en el desarrollo de un videojuego y puede suponer una mala utilización de los recursos disponibles.

La distribución de las distintas etapas entre las tres fases mencionadas anteriormente tampoco está ampliamente consensuado. Predomina la idea de que la fase de Producción agrupa todo aquello que conlleva la obtención de elementos tangibles y elaborados para el juego mientras que la fase de Pre-Producción se asocia con los procesos de obtención de elementos poco tangibles o preliminares, aunque con más propiedad y en el mundo del desarrollo de software, se puede denominar Diseño Conceptual del Juego.

En cualquier caso, cabe destacar que la principal carga de trabajo se sitúa en lo que puede denominarse Diseño General del Juego y en el Diseño Técnico que es donde se aborda fundamentalmente el desarrollo del software del videojuego. Así pues, son estas etapas las que requieren mayor número de recursos y una mayor coordinación entre ellos. La figura 20.11 ilustra un posible planteamiento de organización de fases y etapas extraído de [12].

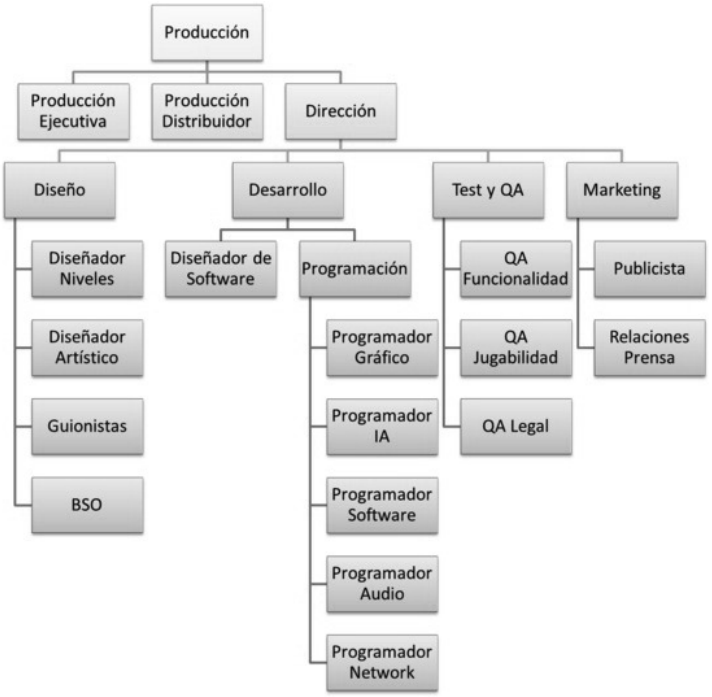


Figura 20.10: Organización de referencia de un equipo de producción de videojuegos

Describimos a continuación cada una de sus etapas de forma más detallada para comprender su objetivo de una forma más clara.

20.2.1. Pre-Producción

En la fase de Pre-Producción se lleva a cabo la concepción de la idea del juego, identificando los elementos fundamentales que lo caracterizarán y finalizando, si es posible, en un diseño conceptual del mismo. Esta información se organiza para dar lugar a lo que puede considerarse una primera versión del documento de diseño del juego o más conocido como GDD (Game Design Document). En este GDD, que debe ser elaborado por el equipo creativo del diseño de videojuegos, se debe identificar y fijar todo lo relacionado con el Diseño del Videojuego que será necesario abordar posteriormente (normalmente en la fase de Producción).

Como patrón de referencia y de acuerdo a lo establecido en [12], el GDD debe contener lo siguiente:

- **Genero**. Clasificación del juego según su naturaleza. La identificación del género al que pertenece el juego servirá para fijar una serie de características básicas para su posterior diseño.
- **Jugadores**. Modalidad de juego: individual o colectivo; multijugador o no; si los jugadores son personas o son máquinas; etc.

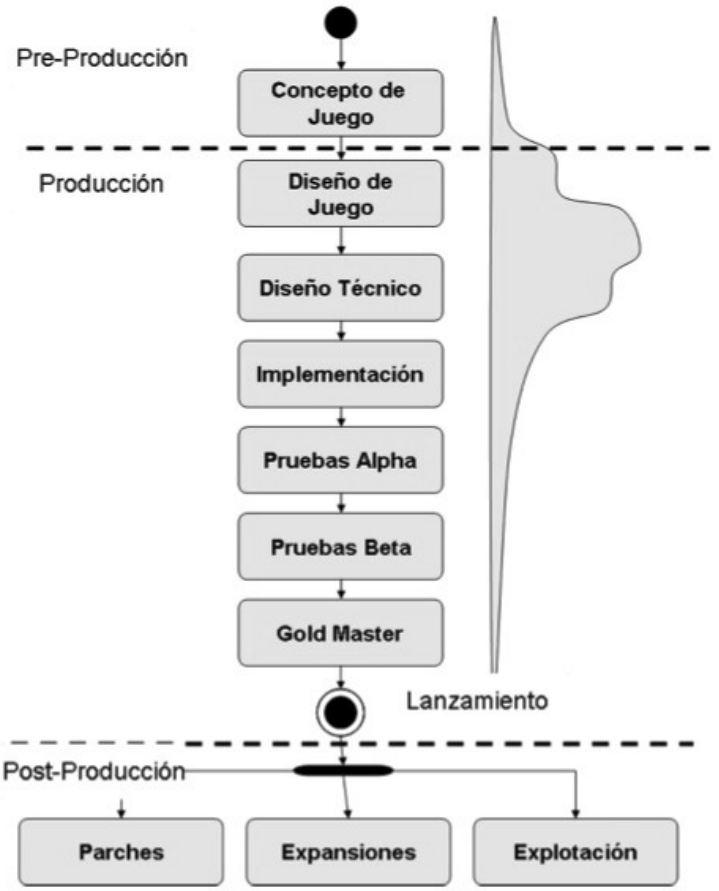


Figura 20.11: Organización de fases y etapas en la producción de un videojuego

- **Historia**. Resumen de la historia del juego. Se realizará una primera aproximación de la trama o la historia a desarrollar durante el juego, destacando qué se quiere contar y cómo se pretende hacerlo. Esto se denomina storyline y storytelling respectivamente.
- **Bocetos**. Los bocetos son diseños preliminares, fundamentalmente, de los personajes y de los escenarios por los que se desarrollará la acción del juego.
- **Look and Feel**. A partir de los bocetos se define el aspecto grafico y artístico del juego, colores, temas dominantes, musicalidad, técnicas de diseño 3D ó 2D, posiciones de cámaras, etc.
- **Interfaz de Usuario**. Se apuntará la forma en la que el jugador interactuará con el juego y con qué mecanismos contará para ello: estilos de interacción, metáforas de interacción, paradigma de interacción, etc.
- **Objetivos**: Se fijan las metas del juego de acuerdo a la historia que se va a desarrollar.
- **Reglas**: Se establece qué acciones podrá desarrollar el jugador y cómo podrá hacerlo.

- **Características**. Se recogen las características principales de cada personaje del juego y de los elementos que intervienen durante su historia.
- **Gameplay**. Este es un concepto poco preciso y de muy amplio alcance, siendo ligeramente diferente su aplicación a cada tipo de juego. En esencia se trata de la naturaleza general del videojuego y de la interactividad que soportará. Es decir, los aspectos fundamentales que caracterizan la forma en la que se va a jugar, las cosas que el jugador va a poder hacer en el juego, la forma en la que el entorno del juego reaccionará a las acciones del jugador, mediadas por los correspondientes personajes, etc. Estos aspectos se describirán sin detallar en exceso a nivel de gráficos, sonido o de la propia historia.
- **Diseño de Niveles**. Se describen los niveles de dificultad que presentará el juego indicando cuántos será y cómo serán, así como los retos a los que el jugador se enfrentará en cada uno de ellos. En algunos casos, estos niveles también pueden estar asociados a etapas o fases del juego.
- **Requerimientos técnicos**. Se definen los requerimientos técnicos de máquina y dispositivos que requerirá el videojuego para su utilización.
- **Marketing**. Esta es una parte esencial en cualquier producto, pero especialmente en el caso de un videojuego todavía más. Muchos videojuegos con fuertes inversiones han sido prácticamente un fracaso por no abordar este aspecto desde las primeras faces de desarrollo. Por lo tanto, es necesario plantear, desde esta fase, la líneas maestras por las que se va a regir la generación de marketing y publicidad del producto.
- **Presupuesto**. Se realizará una primera aproximación al presupuesto que soportará el proyecto de desarrollo del videojuego.

Como se ha indicado anteriormente, esta primera versión del GDD será el punto de partida para iniciar la fase de Producción, pero cabe insistir sobre la importancia de uno de sus elementos: se trata del Gameplay. Dado el carácter un tanto difuso de este concepto, consideremos como ejemplo el caso particular del conocido y clásico juego *"Space Invaders"*. En este juego indicaríamos que se debe poder mover una nave alrededor del cuadrante inferior de la pantalla y disparar a una serie de enemigos que aparecen por la parte superior de la pantalla y que desaparecen cuando son alcanzados por los disparos. Estos enemigos tratan de atacarnos con sus disparos y presionándonos mediante la reducción de nuestro espacio de movimientos e intentando chocar contra nuestra nave.

El Gameplay tiene una implicación importantísima en la calidad final del juego y, por lo extensión, en la Jugabilidad del mismo. Luego los esfuerzos destinados a su análisis y planteamiento revertirán directamente en las propiedades que caracterizan la Juabilidad. No obstante y para profundizar en más detalle sobre este aspecto, se recomienda consultar los siguientes libros: *"Rules of Play: Game Design Fundamentals"* [80] y *"Game Design: Theory and Practice"* [77].

20.2.2. Producción

La fase de Producción es la fase donde se concentra el trabajo principal, en volumen y en número de participantes, del proceso de desarrollo del videojuego, especialmente en lo que se denomina Diseño del Juego y Diseño Técnico. Hay que significar que este curso está orientado, fundamentalmente, a las tareas y técnicas relacionadas con el Diseño Técnico, pero no queremos dejar de situarlo en el contexto del proceso global que requiere llevarse a cabo para concebir y desarrollar un producto de estas características.

Siguiendo lo presentado en la figura 20.11, las etapas principales que se identifican en esta fase son las siguientes:

- Diseño de Juego. Esta es una etapa fundamental en la que se describen con alto nivel de detalle todos los elementos que formarán parte del juego. Principalmente, lo que se hace es refinar lo contemplado en el GDD para obtener su versión definitiva, diseñando en profundidad todos sus aspectos anteriormente especificados. Así se obtiene lo que se denomina DTD (Documento Técnico de Diseño) junto con la Biblia de la Historia, la Biblia del Arte y la primera versión del Motor del Juego. Fundamentalmente, se debe trabajar en tres líneas de trabajo que vienen a caracterizar lo que se denomina diseño del juego y son las siguientes:
  - **Diseño Artístico** que incluye:
    - La Biblia de la Historia donde se recogen todas las historias de los personajes y del mundo donde se desarrolla el juego así como el argumento completo del juego.
    - Biblia del Arte que incluye:
      - Elementos sonoros del juego, es decir, voces, efectos, música, ambiente, etc. Incluso se empieza a trabajar en lo que debe dar lugar al Motor de Sonido.
      - Visualización gráfica de los elementos con los que interactuarán los jugadores.
      - Elementos gráficos como los modelos en 3D, las cámaras, las luces, los sprites, los tiles, etc. De igual manera que en el caso del sonido, esto sirve de punto de partida para comenzar a trabajar en lo que se denomina Motor Gráfico.
  - **Diseño de la Mecánica del Juego**, en el que se trabaja en lo aspectos que se enumeran a continuación:
    - Cómo se va a interactuar en el juego, cuáles son las reglas que lo rigen y cómo es la comunicación que tendrá lugar en caso de tratarse de un juego on-line.
    - Se debe diseñar el comportamiento, habilidades y otros detalles significativos de los personajes y del mundo que les rodea.
    - Se empieza a trabajar en el diseño del motor de IA que pueda requerir y en todo lo asociado con esto.
    - Se diseña lo que se denomina el Motor Físico con el objetivo de generar los aspectos físicos del juego como explosiones, disparos, etc.
  - **Motor del Juego** que hace referencia a una serie de rutinas que permiten la representación de todos los elementos funcionales del juego. En síntesis puede decirse que agrupa todo lo relacionado con el Motor Gráfico, el Motor de Sonido, el Gestor de IA, el Motor Físico y todo el resto de gestores que pueden ser necesario para manejar el universo completo del videojuego.

- **Diseño Técnico**. Ésta se trata de la etapa que directamente está relacionada el desarrollo del software del juego y con lo se aborda en profundidad como contenido técnico esencial de este curso. Es aquí donde de describe cómo será implementado el juego. Para ello se hace uso de notaciones como UML (Unified Modeling Language) y se plantea y decide la metodología de desarrollo software más apropiada según las características y, sobretodo, envergadura del producto software que se pretende implementar. Es importante tener una descripción conceptual y precisa que permita ver el funcionamiento del software

desde puntos de vistas estructurales, dinámicos, de interacción y de despliegue. En definitiva, se trata de un proyecto de desarrollo de software completo que debe incluir también una planificación de tareas a realizar, una asignación a los miembros del equipo de desarrolladores. Esto incluye la identificación de hitos importantes, las fechas de entrega y el análisis de riesgos.

- **Implementación**. En esta etapa debe abordarse la implementación de los elementos software del proyecto que se describieron en la etapa anterior, utilizando para ello métodos, técnicas y herramientas como las que se trabajan a lo largo de este curso. Es posible que se detecten algunos errores del diseño inicial y que se requieran revisiones. En muchos casos, esta etapa y la anterior son repetidas de forma iterativa o se someten a ciclos iterativos. Esto, en muchos casos viene determinado por la metodología de desarrollo software que se emplea y que, como se ha apuntado anteriormente, depende de muchos factores como la envergadura del proyecto, los recursos disponibles, etc. Generalmente, en este momento se suelen construir demos reducidas del juego que son objeto de publicación, contribuyendo así a materializar la campaña de marketing y publicidad que tan esenciar es para lograr el éxito comercial del producto.

- **Pruebas Alpha**. Estas pruebas se abordan cuando tenemos ya partes del producto software terminado. También se suelen denominan pruebas Code Complete. Mediante las mismas, el producto se somete a diversas pruebas que realizan pequeños equipos que han estado llevando a cabo el proceso de diseño y desarrollo del juego. El objetivo de las mismas es buscar pequeños errores y refinar algunos aspectos. Uno de los aspectos más importantes que se valoran en esta etapa es la Jugabilidad del juego a través de diversas propiedades y facetas como se describió anteriormente.

- **Pruebas Beta**. En las pruebas Beta o también denominadas Content Complete se finaliza todo lo relacionado con contenidos como el decorado de las misiones, los gráficos, los textos en diferentes idiomas, doblaje del sonido, etc. Además, se trabaja para asegurar que los contenidos incluidos en el juego se ajustan a las leyes vigentes y a la ética establecida en aquellos países donde se pretende comercializar el juego. Estas pruebas son llevadas a cabo por personas ajenas al equipo de desarrollo.

- **Gold Master**. Esta etapa aborda una prueba definitiva con el producto final que se publicará y que se producirá. Obviamente, incluye todo el contenido artístico, técnico y documental (es decir, los manuales de usuario). En este momento, la publicidad deber ser la mayor posible, incluyéndose la realización de reportajes, artículos, etc.

20.2.3. Post-Producción

La fase de Post-Producción, en la que no nos vamos a detener ya que se aleja bastante del contenido tratado en el curso, aborda fundamentalmente la explotación y el mantenimiento del juego como si de cualquier otro producto software se tratase.

20.3. Metodologías Alternativas

El método descrito anteriormente prácticamente es un caso particular de aplicación del Modelo de Proceso en Cascada, que conlleva la finalización de una etapa antes de poder abordar la siguiente. En el caso del desarrollo de software, esto condiciona bastante lo relacionado con las etapas de pruebas, cuya realización se retrasa en exceso quedando situada casi al final del desarrollo. En ese momento, depurar y solucionar cualquier problema, si es que es posible, puede resultar excesivamente costoso en tiempo y, en consecuencia, en dinero.

Precisamente, en el área del desarrollo de sistemas interactivos, está claramente establecido que las pruebas, sobretodo de Usabilidad, deben hacerse desde las primeras fases, incluso cuando los prototipos están únicamente a nivel de bocetos y en papel. Así pues, eso entra firmemente en contradicción con el hecho de que un videojuego se considere como un caso particular de sistema interactivo.

Por otro lado, la necesidad de evaluar lo antes posible las propiedades relacionadas con la Jugabilidad y la Experiencia del Jugador requieren plantear variaciones a la metodología de producción y desarrollo anteriormente presentada. Por esta razón, se describen a continuación algunos otros métodos alternativos que se utilizan en la industria del desarrollo de software de videojuegos.

20.3.1. Proceso Unificado del Juego

Tomando como punto de partida el PUD (Proceso Unificado de Desarrollo) de IBM, en [35] se plantea la metodología denominada Proceso Unificado del Juego (o GUP (Game Unified Process)). Este método se caracteriza por incentivar la comunicación entre los equipos de trabajo que abordan cada etapa del desarrollo, la documentación estricta de cada paso y por abordar el proceso de desarrollo de una forma iterativa y en ciclos muy cortos. Se puede considerar como una versión ágil de la metodología PUD particularizada para el desarrollo de software de videojuegos.

Además, este método propone la utilización del paradigma de Programación Extrema [11] como instrumento para agilizar el desarrollo del software del videojuego. Por lo tanto, esto es especialmente aplicable a lo que serían las etapas de Diseño del Juego, Diseño Técnico, Implementación y Pruebas.

20.3.2. Desarrollo Incremental

Otro método que puede ser adecuado, si se pretende potenciar la realización de pruebas en las fases más tempranas y obtener la correspondiente realimentación, es el *Desarrollo Incremental* de Sikora [88]. Básicamente, se introduce la idea de disponer de un equipo de "jugadores" dentro del equipo de desarrolladores encargados de las pruebas. Estos "jugadores" siempre realizan una subetapa de pruebas en cada etapa antes de validar los resultados y poder asumir las tareas de la siguiente etapa.

20.3.3. Desarrollo Ágil y Scrum

Una de las metodologías que mejores resultados está produciendo recientemente en la industria del software de videojuegos es la propuesta por Clinton Keith dentro de su estudio de desarrollo *High Moon* [54]. Como ejemplo de caso de éxito en el que se ha aplicado esta metodología, cabe mencionar *DarkWatch*.

Esta metodología plantea la utilización de procesos ágiles de desarrollo de software, unido a los pilares básico de la metodología de desarrollo de productos Scrum [95].

El objetivo principal del método de Keith es hacer un diseño centrado en el jugador y en los resultados del proceso de desarrollo en cada una de sus fases. Así, se resalta la importancia de obtener la opinión del usuario en cada momento, por lo que intenta involucrar al equipo de pruebas lo antes posible. De esta forma, se facilitará la posibilidad detectar y solucionar a tiempo todos los posibles errores y se podrá analizar la Juabilidad en cada momento para ir mejorándola continuamente, del mismo modo que se hace para el caso particular de la Usabilidad en un sistema interactivo.

Esta metodología requiere de la realización de importantes esfuerzos iniciales para lograr obtener prototipos básicos pero jugables y, por lo tanto, evaluables. Con estos prototipos se inicia un proceso iterativo en el que el equipo de pruebas lo utiliza y proporciona realimentación orientada a la mejora, especialmente de la Jugabilidad pero también de otros detalles que pueden caracterizar el producto final.

Información mucho más detallada de cómo aplicar esta metodología puede encontrarse en el libro *"Agile Game Development with Scrum"* de [55].

20.3.4. Desarrollo Centrado en el Jugador

En esta subsección se va a describir la propuesta de [40] que está inspirada directamente en los principios fundamentales del DCU (Diseño Centrado en el Usuario) y de las metodologías de desarrollo software que se han derivado de los mismos.

La idea fundamental del DCU, como ya se ha apuntado anteriormente, es la involucrar al usuario y hacerlo al principio de cualquier proceso de desarrollo, ya que muchos de los problemas del software se deben a una carencia en las fases iniciales del desarrollo, concretamente en las fases de elicitación y de análisis de requisitos. Esto ya ha sido contemplado en diversos estándares que plantean ciclos de vida del proceso que incluyen modelos de madurez para la Usabilidad como pilar fundamental que garantizar el éxito del producto en cuanto a la Experiencia del Usuario.

De la misma forma que el DCU es necesario para el desarrollo de aplicaciones que cubran los requisitos del usuario de forma adecuada, el Diseño Centrado en el Jugador es especialmente importante para considerar la diversidad y subjetividad de los perfiles de jugadores existentes. Además, esto contribuye directamente a la reducción de la proliferación de productos que requieren numerosos "parches" incluso desde los primeros meses de vida en el mercado.

En este sentido [40] propone un método inspirado directamente en la metodología *PPIu+a* propuesta en [41] para Ingeniería de la Usabilidad y que se resume en la figura 20.12. Para facilitar su comprensión puede utilizarse la figura 20.13 en la que se relaciona y compara esta metodología MPIu+a.

En las fuentes citadas pueden encontrar muchos más detalles sobre la fases más destacables que son las de análisis, diseño, desarrollo y evaluación de elementos jugables. Especialmente, se plantea un patrón a seguir para la obtención de requisitos de Jugabilidad con ejemplos de aplicación, se proponen una serie de guías de estilo para llevar a cabo un diseño que fomente la Jugabilidad, se muestra cómo aplicar Scrum y programación extrema para la construcción de prototipos jugables y se describe cómo evaluar la Jugabilidad de los prototipos para obtener conclusiones sobre la experiencia del jugador.

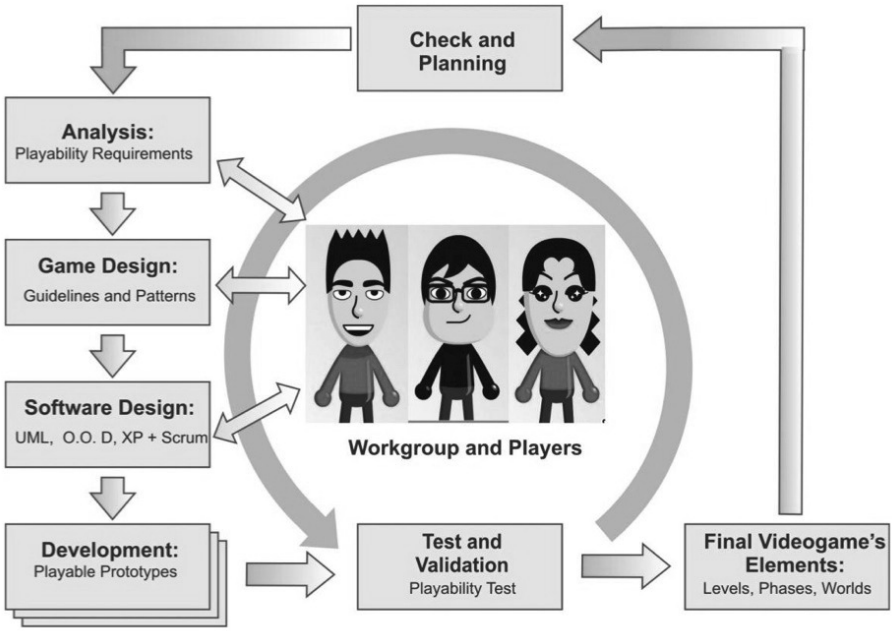


Figura 20.12: Método de Diseño Centrado en el Jugador de [40]

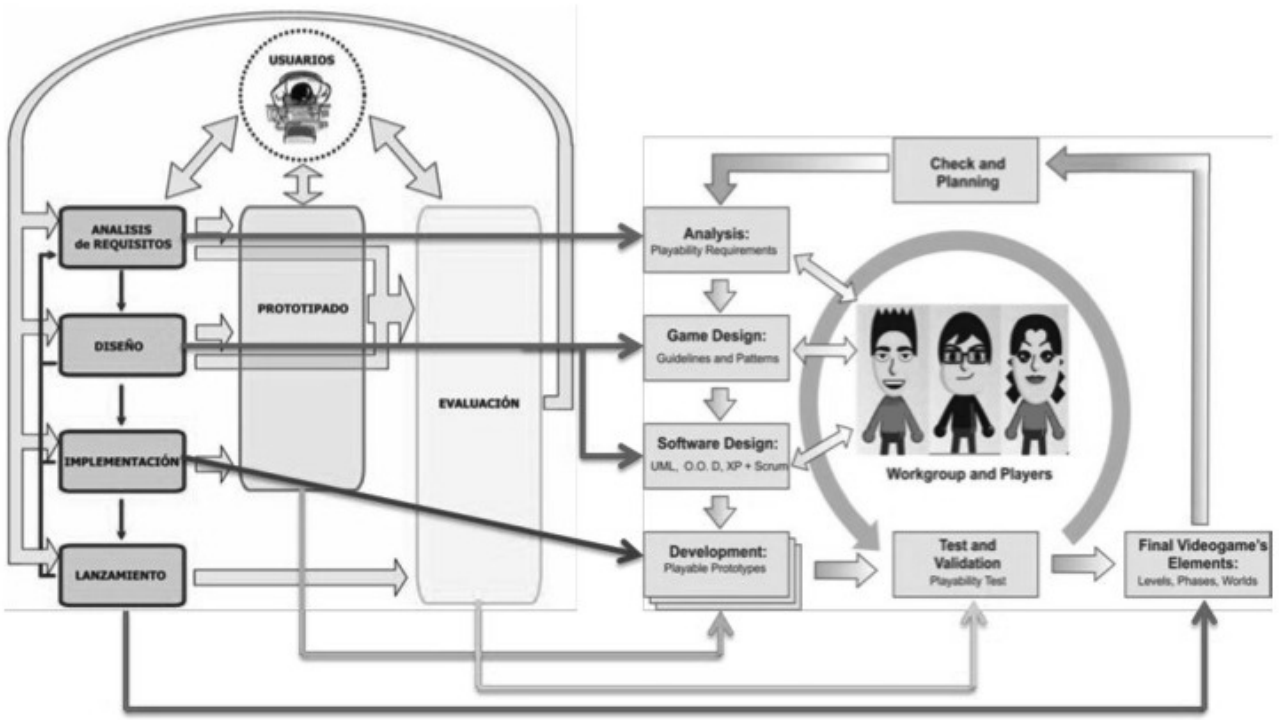


Figura 20.13: Comparación entre el método de Diseño Centrado en el Jugador y el de Diseño Centrado en el Usuario de MPIu+a

Capítulo 21
C++ Avanzado

Francisco Moya Fernández
David Villa Alises
Sergio Pérez Camacho
Cleto Martín Angelina

21.1. Estructuras de datos no lineales

Cualquier programa donde la eficiencia sea importante, y es el caso de la mayoría de los videojuegos, necesitan estructuras de datos específicas. Hay varios motivos para ello:

- Hasta ahora hemos estudiado fundamentalmente la STL, que oculta la estructura real de los contenedores ofreciendo un aspecto de estructura lineal. Así, por ejemplo, los objetos de tipo `map` o `set` se representan realmente mediante árboles, aunque el programador está completamente aislado de ese detalle de implementación. Solo podemos anticipar la estructura subyacente mediante indicadores indirectos, como la complejidad de las operaciones o la estabilidad de los iteradores.
- Algunas estructuras, como es el caso de los grafos, no tienen una representación lineal evidente y se pueden recorrer de distintas formas. Por tanto debe existir un número variable de iteradores.
- Las estructuras de la STL están diseñadas para uso general. El diseñador no puede anticipar en qué condiciones se van a usar por lo que toma las decisiones apropiadas para el mayor número de casos posible. Conociendo los detalles (gestión de memoria, algoritmos que se van a aplicar) se pueden obtener rendimientos muy superiores con mínimas modificaciones sobre la estructura subyacente.

Como ya hemos puntualizado en capítulos anteriores, es muy importante no optimizar de manera prematura. Para ilustrar este aspecto veamos el siguiente ejemplo tomado de [17], capítulo 11.

Anti-optimizaciones

Con los compiladores actuales es muy difícil implementar código equivalente a la STL más eficiente. Algunos ejemplos de [17] hoy en día son completamente diferentes.

Listado 21.1: Dos formas de sumar enteros

```
1  int myIntegerSum(int* a, int size) {
2      int sum=0;
3      int* begin = a;
4      int* end = a + size;
5      for (int* p = begin; p != end; ++p)
6      sum += *p;
7      return sum;
8  }
9
10 int stlIntegerSum(int* a, int size) {
11     return accumulate(a, a+size, 0);
12 }
```

En dicho libro se argumentaba que la función `myIntegerSum()` es casi cuatro veces más rápida que `stlIntegerSum()`. Y probablemente era verdad en el año 1999. Sin embargo hoy en día, empleando GNU g++ 4.6.2 o clang++ 3.0 el resultado es prácticamente idéntico, con una muy ligera ventaja hacia la versión basada en la STL.

21.1.1. Árboles binarios

Las estructuras arborescentes se encuentran entre las más utilizadas en la programación de todo tipo de aplicaciones. Ya hemos visto en el módulo 2 algunas de sus aplicaciones para el mezclado de animaciones (*Priority Blend Tree*), o para indexar el espacio (*BSP* Tree, quatree, octree, BBT). Estudiaremos su funcionamiento en este capítulo, pero el desarrollo de videojuegos no se limita a los gráficos, por lo que otro tipo de árboles más generales pueden resultar también necesarios.

Los árboles se utilizan con frecuencia como mecanismo eficiente de búsqueda. Para este fin implementan un rico conjunto de operaciones: búsqueda de un elemento, mínimo o máximo, predecesor o sucesor de un elemento y las clásicas operaciones de inserción y borrado. Se pueden emplear como un diccionario o como una cola con prioridad.

Todas estas operaciones están presentes en los contenedores ordenados de la STL, singularmente `set`, `multiset`, `map` y `multimap`. No debe extrañar por tanto que en todos ellos se emplea una variante de árbol binario denominada *red-black tree*.

Un nodo de árbol contiene habitualmente un atributo `key` que se emplea para compararlo con otros nodos y además mantiene un conjunto de punteros a otros nodos que mantienen su relación con el resto de la estructura. Así, por ejemplo, los nodos de árboles binarios mantienen un atributo `parent` que apunta al nodo padre, y un par de punteros *left* y *right* que apuntan al hijo por la izquierda y por la derecha respectivamente. A su vez cada hijo puede tener otros nodos hijos, por lo que realmente cada nodo cuenta con dos subárboles (izquierdo y derecho).

Las operaciones básicas de los árboles se ejecutan en un tiempo proporcional a la altura del árbol. Eso implica $O(\log n)$ en el caso peor si está correctamente balanceado, pero $O(n)$ si no lo está.

Árboles de búsqueda binaria

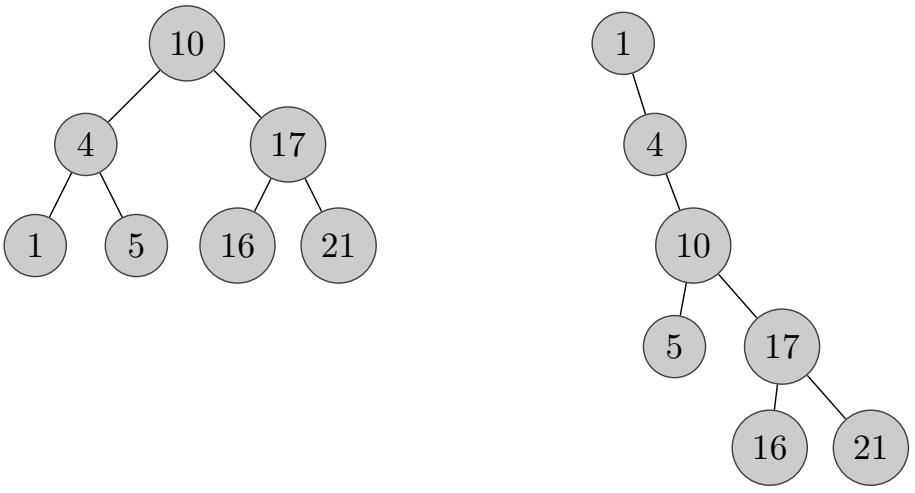


Figura 21.1: Dos árboles de búsqueda binaria. Ambos contienen los mismos elementos pero el de la izquierda es mucho más eficiente

Los árboles de búsqueda binaria se definen por la siguiente propiedad:

> Todos los nodos del subárbol izquierdo de un nodo tienen una clave menor o igual a la de dicho nodo. Análogamente, la clave de un nodo es siempre menor o igual que la de cualquier otro nodo del subárbol derecho.

Por tratarse del primer tipo de árboles expondremos con cierto detalle su implementación. Como en cualquier árbol necesitamos modelar los nodos del árbol, que corresponden a una simple estructura:

Listado 21.2: Estructura de un nodo de árbol de búsqueda binaria

```
1 template <typename KeyType>
2 struct Node {
3     typedef Node<KeyType> NodeType;
4
5     KeyType key;
6     NodeType* parent;
7     NodeType* left;
8     NodeType* right;
```

Sobre esta misma estructura es posible definir la mayoría de las operaciones de un árbol. Por ejemplo, el elemento más pequeño podría definirse como un método estático de esta manera:

Listado 21.3: Búsqueda del elemento mínimo en un árbol de búsqueda binaria

```
1     static NodeType* minimum(NodeType* x) {
2         if (x == 0) return x;
3         if (x->left != 0) return minimum(x->left);
4         return x;
5     }
```

Para obtener el mínimo basta recorrer todos los subárboles de la izquierda y análogamente para encontrar el máximo hay que recorrer todos los subárboles de la derecha hasta llegar a un nodo sin subárbol derecho.

Listado 21.4: Búsqueda del elemento máximo en un árbol de búsqueda binaria

```
1     static NodeType* maximum(NodeType* x) {
2         if (x == 0) return x;
3         if (x->right != 0) return maximum(x->right);
4         return x;
5     }
```

El motivo de utilizar métodos estáticos en lugar de métodos normales es poder invocarlos para el nodo nulo. Los métodos de clase invocados sobre un objeto nulo tienen un comportamiento indefinido.

Nuestra implementación del método estático `minimum()` es recursiva. Con frecuencia se argumenta que una implementación iterativa es más eficiente porque no crea un número indefinido de marcos de pila. Realmente eso depende del tipo de recursión. Cuando el compilador puede detectar recursión por la cola, es decir, cuando tras la llamada recursiva no quedan operaciones pendientes de realizar, el compilador puede optimizar el código y eliminar completamente la llamada recursiva.

Las instancias de `Node` no tienen por qué ser visibles directamente al programador, al igual que los contenedores tipo `set` de la STL. Por ejemplo, esto puede lograrse utilizando un *namespace* privado.

La búsqueda de un elemento también puede plantearse con un algoritmo recursivo aprovechando la propiedad que define a los árboles de búsqueda binaria:

Listado 21.5: Búsqueda de una clave en un árbol de búsqueda binaria

```
1     static NodeType* search(NodeType* x, KeyType k) {
2         if (x == 0 || x->key == k) return x;
3         else if (k < x->key) return search(x->left, k);
4         else return search(x->right, k);
5     }
```

También pueden implementarse de manera directa los métodos `successor()` y `predecesor()` para encontrar el nodo siguiente o anterior a uno dado según el orden de las claves:

Listado 21.6: Búsqueda del sucesor de un nodo en un árbol de búsqueda binaria

```
1     static NodeType* successor(NodeType* x) {
2         if (x->right != 0) return minimum(x->right);
3         NodeType* parent = x->parent;
4         while (parent != 0 && x == parent->right) {
5             x = parent;
6             parent = x->parent;
7         }
8     }
```

Si hay un subárbol a la derecha del nodo entonces es el mínimo de ese subárbol (en la figura 21.1 izquierda el sucesor de 10 es 16). Si no lo hay entonces tendremos que subir hasta el primer padre que tiene al nodo como subárbol izquierdo (en la figura 21.1 izquierda el sucesor de 5 es 10).

Se propone como ejercicio la implementación de la búsqueda del predecesor de un nodo determinado.

El resto de las operaciones básicas sobre un árbol (inserción y borrado de elementos) requiere de una estructura que actúa como fachada de los nodos del árbol.

Listado 21.7: Estructura de un árbol de búsqueda binaria

```
1 template <class KeyType>
2 struct Tree {
3     typedef Node<KeyType> NodeType;
4
5     NodeType* root;
```

El atributo `root` mantiene cuál es el nodo raíz del árbol. Los métodos de inserción y borrado deben actualizarlo adecuadamente.

Listado 21.8: Inserción en un árbol de búsqueda binaria

```
1     void insert(NodeType* z) {
2         NodeType* y = 0;
3         NodeType* x = root;
4         while (x != 0) {
5             y = x;
6             if (z->key < x->key)
7                 x = x->left;
8             else
9                 x = x->right;
10        }
11        z->parent = y;
12        if (y == 0) root = z;
13        else if (z->key < y->key)
14            y->left = z;
15        else
16            y->right = z;
17    }
```

Básicamente replica el procedimiento de búsqueda para encontrar el hueco donde debe insertar el elemento, manteniendo el padre del elemento actual para poder recuperar el punto adecuado al llegar a un nodo nulo.

El procedimiento más complejo es el de borrado de un nodo. De acuerdo a [20] se identifican los cuatro casos que muestra la figura 21.2. Un caso no representado es el caso trivial en el que el nodo a borrar no tenga hijos. En ese caso basta con modificar el nodo padre para que el hijo correspondiente sea el objeto nulo. Los dos primeros casos de la figura corresponden al borrado de un nodo con un solo hijo, en cuyo caso el hijo pasa a ocupar el lugar del nodo a borrar. El tercer caso corresponde al caso en que el hijo derecho no tenga hijo izquierdo o el hijo izquierdo no tenga hijo derecho, en cuyo caso se puede realizar la misma operación que en los casos anteriores enlazando adecuadamente las dos ramas. El cuarto caso corresponde al caso general, con dos hijos no nulos. En ese caso buscamos un sucesor del subárbol izquierdo que no tenga hijo izquierdo, que pasa a reemplazar al nodo, reajustando el resto para mantener la condición de árbol de búsqueda binaria.

Con el objetivo de facilitar el movimiento de subárboles definimos el método `transplant()`. El subárbol con raíz `u` se reemplaza con el subárbol con raíz `v`.

Listado 21.9: Transplantado de subárboles en un árbol de búsqueda binaria

```
1     void transplant(NodeType* u, NodeType* v) {
2         if (u->parent == 0)
3             root = v;
4         else if (u == u->parent->left)
5             u->parent->left = v;
6         else
7             u->parent->right = v;
8         if (v != 0)
9             v->parent = u->parent;
10    }
```

Nótese que no alteramos el nodo padre de `v` ni los hijos de `v`. La responsabilidad de actualizarlos corresponde al que llama a `transplant()`.

Empleando este procedimiento auxiliar es muy sencilla la implementación de `remove()`.

Listado 21.10: Borrado en un árbol de búsqueda binaria

```
1     void remove(NodeType* z) {
2         if (z->left == 0)
3             transplant(z, z->right);
4         else if (z->right == 0)
5             transplant(z, z->left);
6         else {
7             NodeType* y = NodeType::minimum(z->right);
8             if (y->parent != z) {
9                 transplant(y, y->right);
10                y->right = z->right;
11                y->right->parent = y;
12            }
13            transplant(z, y);
14            y->left = z->left;
15            y->left->parent = y;
16        }
17    }
```

Todos los procedimientos básicos (`minimum()`, `maximum()`, `search()`, `predecesor()`, `successor()`, `insert()` y `remove()`) se ejecutan en tiempo $O(h)$ donde h es la altura del árbol. Si el árbol está equilibrado esto implica $O(\log n)$.

Red-black trees

La eficiencia de un árbol de búsqueda binaria depende enormemente del orden en que se introduzcan los elementos. Pueden ser muy eficientes o en el caso peor degenerar a una simple lista doblemente enlazada. Para resolver este problema se han propuesto multitud de esquemas que garantizan que el árbol siempre está equilibrado complicando ligeramente la inserción y borrado.

Los árboles rojo-negro son un caso de estos árboles de búsqueda binaria balanceados. Cada nodo almacena un bit extra, el color, que puede ser rojo o negro. En cada camino simple desde el nodo raíz a una hoja se restringen los colores de manera que nunca pueda ser un camino más del doble de largo que otro cualquiera:

1. Cada nodo es rojo o negro.
2. El nodo raíz es negro.
3. Las hojas del árbol (objetos nulos) son negras.

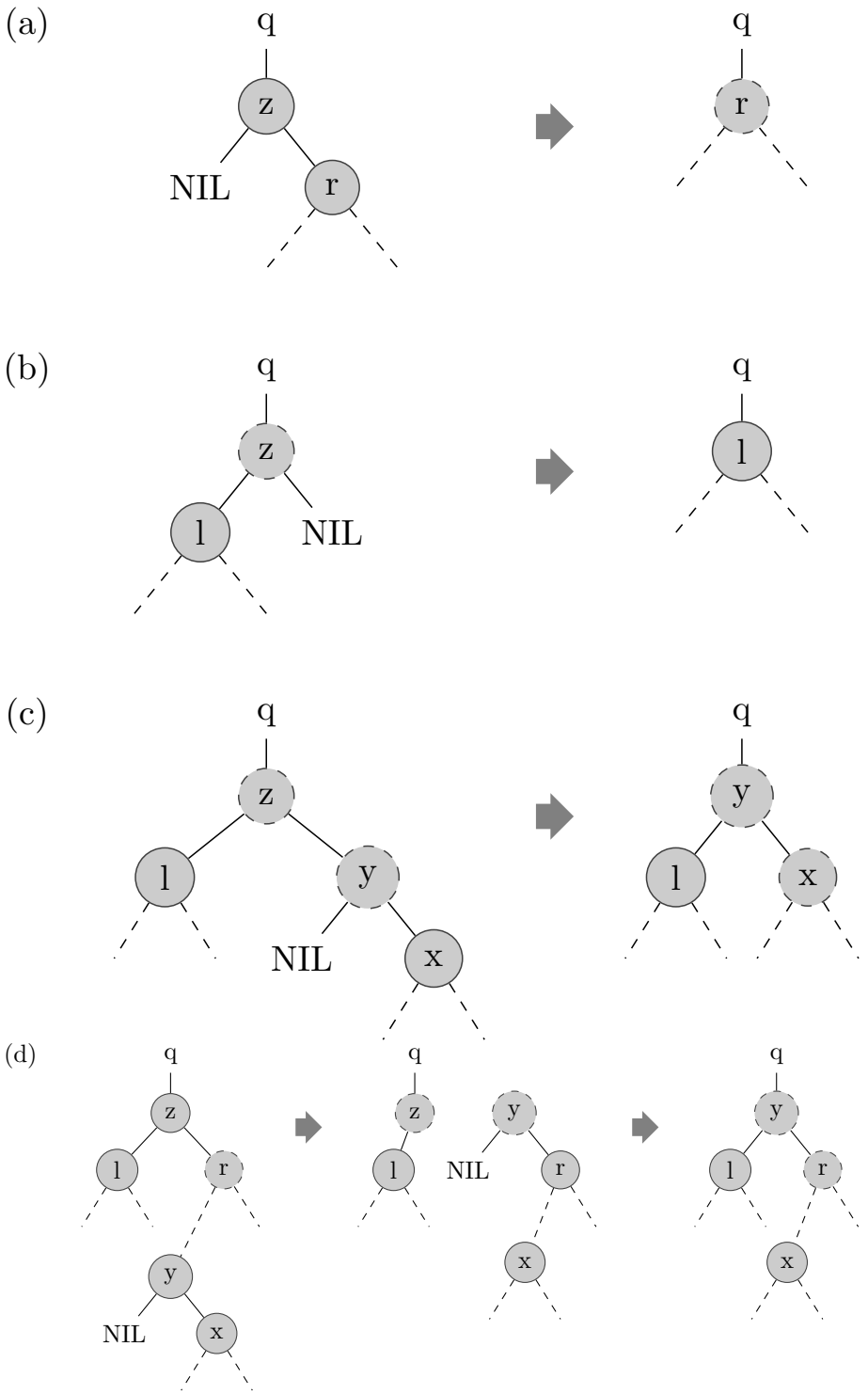


Figura 21.2: Casos posibles según [20] en el borrado de un nodo en un árbol de búsqueda binaria

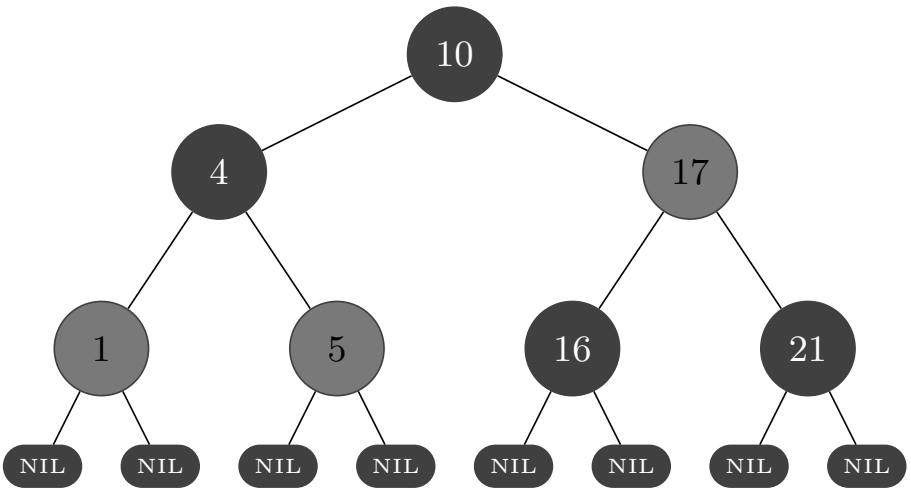


Figura 21.3: Ejemplo de árbol rojo-negro. Los nodos hoja no se representarán en el resto del texto.

4. Los hijos de un nodo rojo son negros.
5. Los caminos desde un nodo a todas sus hojas descendientes contienen el mismo número de nodos negros.

Podemos simplificar los algoritmos eliminando la necesidad de comprobar si es un nodo nulo antes de indexar un elemento sin más que utilizar un nodo especial que usamos como centinela. La estructura del nodo podría ser algo así:

Listado 21.11: Definición de un nodo de un árbol rojo-negro.

```
template <typename KeyType>
struct Node {
    typedef Node<KeyType> NodeType;
    enum Color { Red = false, Black = true };

    KeyType key;
    NodeType* parent;
    NodeType* left;
    NodeType* right;
    Color color;

    Node() {
        left = right = parent = nil();
    }

    static NodeType* nil() {
        if (!_nil)
            _nil = new Node(Black);
        return &_nil;
    }
```

Las operaciones `maximum()`, `minimum()`, `search()`, `successor()` y `predecesor()` son completamente análogas a las de los árboles de búsqueda binaria tradicionales, salvo que ahora está garantizado que se ejecutan en tiempo $O(\log n)$. Por ejemplo, la función `maximum()` sería:

Listado 21.12: Búsqueda del mayor elemento en un árbol rojo-negro.

```
1     static NodeType* maximum(NodeType* x) {
2         if (x->right != NodeType::nil()) return maximum(x->right);
3         return x;
4     }
```

Nótese que ya no es necesario comprobar si `x` es nulo antes de indexar su miembro `right`, puesto que para representar al nodo nulo usamos un centinela perfectamente válido.

En cambio las operaciones de inserción y borrado deben ser modificadas para garantizar que se mantienen las propiedades de árbol rojo-negro. Para ello nos apoyaremos en dos funciones auxiliares: `rotate_left()` y `rotate_right()`:

Listado 21.13: Rotación a la izquierda en un árbol de búsqueda binaria

```
1     void rotate_left(NodeType* x) {
2         NodeType* y = x->right;
3         x->right = y->left;
4         if (y->left != NodeType::nil())
5             y->left->parent = x;
6         y->parent = x->parent;
7         if (x->parent == NodeType::nil())
8             root = y;
9         else if (x == x->parent->left)
10            x->parent->left = y;
11        else
12            x->parent->right = y;
13        y->left = x;
14        x->parent = y;
15    }
```

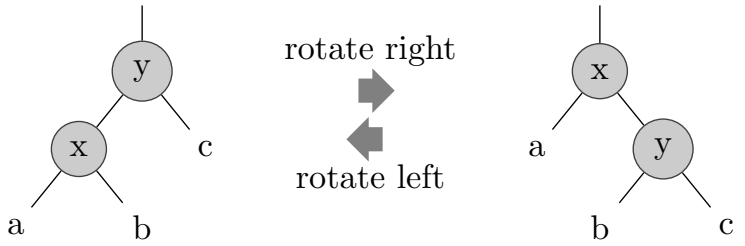


Figura 21.4: Operación de rotación a la derecha o a la izquierda en un árbol de búsqueda binaria

La operación dual `rotate_right()` puede implementarse simplemente intercambiando en el algoritmo anterior `x` por `y`, y `left` por `right`.

La inserción puede ahora realizarse de una forma muy parecida al caso general asumiendo que el color del nodo a insertar es rojo y después arreglando el árbol con rotaciones y cambios de color.

Listado 21.14: Inserción en un árbol rojo-negro

```
1     void insert(NodeType* z) {
2         NodeType* y = NodeType::nil();
3         NodeType* x = root;
4         while (x != NodeType::nil()) {
5             y = x;
6             if (z->key < x->key)
7                 x = x->left;
8             else
```

```
9                   x = x->right;
10              }
11              z->parent = y;
12              if (y == NodeType::nil())
13                  root = z;
14              else if (z->key < y->key)
15                  y->left = z;
16              else
17                  y->right = z;
18              z->left = NodeType::nil();
19              z->right = NodeType::nil();
20              z->color = Node::Color::Red;
21              insert_fixup(z);
22          }
```

Al asumir el color rojo podemos haber violado las reglas de los árboles rojo-negro. Por esta razón llamamos a la función `insert_fixup()` que garantiza el cumplimiento de las reglas tras la inserción:

Listado 21.15: Reparación tras la inserción en árbol rojo-negro

```
1       void insert_fixup(NodeType* z) {
2           while (z->parent->color == Node::Color::Red) {
3               if (z->parent == z->parent->parent->left) {
4                   NodeType* y = z->parent->parent->right;
5                   if (y->color == Node::Color::Red) {
6                       z->parent->color = Node::Color::Black;
7                       y->color = Node::Color::Black;
8                       z->parent->parent->color = Node::Color::Red;
9                       z = z->parent->parent;
10                  }
11                  else {
12                      if (z == z->parent->right) {
13                          z = z->parent;
14                          rotate_left(z);
15                      }
16                      z->parent->color = Node::Color::Black;
17                      z->parent->parent->color = Node::Color::Red;
18                      rotate_right(z->parent->parent);
19                  }
20              }
21              else {
22                  NodeType* y = z->parent->parent->left;
23                  if (y->color == Node::Color::Red) {
24                      z->parent->color = Node::Color::Black;
25                      y->color = Node::Color::Black;
26                      z->parent->parent->color = Node::Color::Red;
27                      z = z->parent->parent;
28                  }
29                  else {
30                      if (z == z->parent->left) {
31                          z = z->parent;
32                          rotate_right(z);
33                      }
34                      z->parent->color = Node::Color::Black;
35                      z->parent->parent->color = Node::Color::Red;
36                      rotate_left(z->parent->parent);
37                  }
38              }
39          }
40          root->color = Node::Color::Black;
41      }
```

La inserción de un nodo rojo puede violar la regla 2 (el nodo raíz queda como rojo en el caso de un árbol vacío) o la regla 4 (el nodo insertado pasa a ser hijo de un nodo rojo). Este último caso es el que se contempla en el bucle de la función `insert_fixup()`. Cada una de las dos ramas del *if* sigue la estrategia dual, dependiendo de si el padre es un hijo derecho o izquierdo. Basta estudiar el funcionamiento de una rama, dado que la otra es idéntica pero intercambiando `right` y `left`. Básicamente se identifican tres casos.

- El primero corresponde a las líneas [6] a [9]. Es el caso en que el nodo a insertar pasa a ser hijo de un nodo rojo cuyo hermano también es rojo (e.g. figura 21.5.a). En este caso el nodo padre y el nodo tío se pasan a negro mientras que el abuelo se pasa a rojo (para mantener el número de nodos negros en todos los caminos). Al cambiar a rojo el nodo abuelo es posible que se haya vuelto a violar alguna regla, y por eso se vuelven a comprobar los casos.
- Otra posibilidad es que el nodo tío sea negro y además el nodo insertado sea hijo derecho (e.g. figura 21.5.b). En ese caso se realiza una rotación a la izquierda para reducirlo al caso siguiente y se aplica lo correspondiente al último caso.
- El último caso corresponde a que el nodo tío sea negro y el nodo insertado sea hijo izquierdo (e.g. figura 21.5.c). En ese caso se colorea el padre como negro, y el abuelo como rojo, y se rota a la derecha el abuelo. Este método deja un árbol correcto.

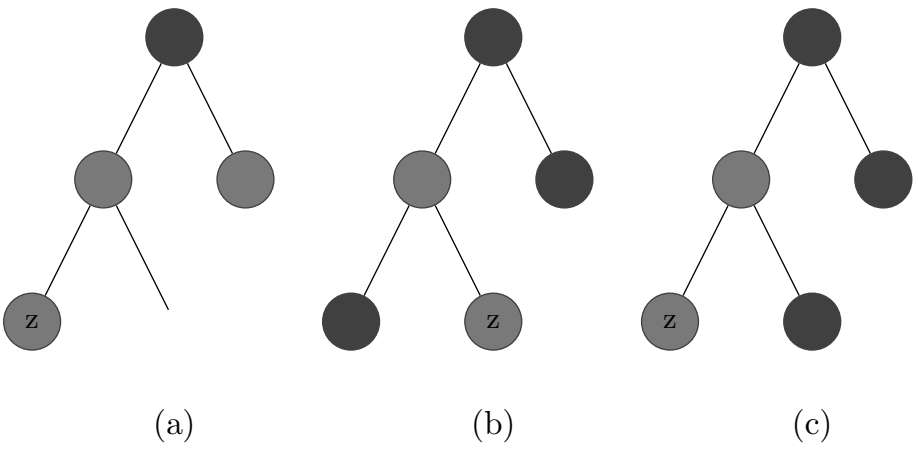


Figura 21.5: Casos contemplados en la función `insert_fixup()`.

El borrado también se apoya en la función `transplant()` que es muy similar al caso de los árboles de búsqueda binaria.

Listado 21.16: Transplantado de subárboles en árbol rojo-negro

```
1     void transplant(NodeType* u, NodeType* v) {
2         if (u->parent == NodeType::nil())
3             root = v;
4         else if (u == u->parent->left)
5             u->parent->left = v;
6         else
7             u->parent->right = v;
8         v->parent = u->parent;
9     }
```

Con este procedimiento auxiliar el borrado de un nodo queda relativamente similar al caso de árboles de búsqueda binaria.

Listado 21.17: Borrado de un nodo en árboles rojo-negro

```
1     void remove(NodeType* z) {
2         NodeType* y = z;
3         NodeType* x;
4         NodeType::Color y_orig_color = y->color;
5         if (z->left == NodeType::nil()) {
6             x = z->right;
7             transplant(z, z->right);
8         }
9         else if (z->right == NodeType::nil()) {
10            x = z->left;
11            transplant(z, z->left);
12        }
13        else {
14            y = Node::minimum(z->right);
15            y_orig_color = y->color;
16            x = y->right;
17            if (y->parent == z) {
18                x->parent = y;
19            }
20            else {
21                transplant(y, y->right);
22                y->right = z->right;
23                y->right->parent = y;
24            }
25            transplant(z, y);
26            y->left = z->left;
27            y->left->parent = y;
28            y->color = z->color;
29        }
30        if (y_orig_color == Node::Color::Black)
31            rb_remove_fixup(x);
32    }
```

El nodo `y` corresponde al nodo que va a eliminarse o moverse dentro del árbol. Será el propio `z` si tiene menos de dos hijos o el nodo *y* de los casos c y d en la figura 21.2. Mantenemos la variable `y_orig_color` con el color que tenía ese nodo que se ha eliminado o movido dentro del árbol. Solo si es negro puede plantear problemas de violación de reglas, porque el número de nodos negros por cada rama puede variar. Para arreglar los problemas potenciales se utiliza una función análoga a la utilizada en la inserción de nuevos nodos.

Listado 21.18: Reparación tras borrar un nodo en árboles rojo-negro

```
1     void remove_fixup(NodeType* x) {
2         while (x != root && x->color == Node::Color::Black) {
3             if (x == x->parent->left) {
4                 NodeType* w = x->parent->right;
5                 if (w->color == Node::Color::Red) {
6                     w->color = Node::Color::Black;
7                     x->parent->color = Node::Color::Red;
8                     rotate_left(x->parent);
9                     w = x->parent->right;
10                }
11                if (w->left->color == Node::Color::Black
12                    && w->right->color == Node::Color::Black) {
13                    w->color = Node::Color::Red;
14                    x = x->parent;
15                }
16                else {
17                    if (w->right->color == Node::Color::Black) {
18                        w->left->color = Node::Color::Black;
```

```
19                          w->color = Node::Color::Red;
20                          rotate_right(w);
21                          w = x->parent->right;
22                      }
23                      w->color = x->parent->color;
24                      x->parent->color = Node::Color::Black;
25                      w->right->color = Node::Color::Black;
26                      rotate_left(x->parent);
27                      x = root;
28                  }
29              }
30              else {
31                  NodeType* w = x->parent->left;
32                  if (w->color == Node::Color::Red) {
33                      w->color = Node::Color::Black;
34                      x->parent->color = Node::Color::Red;
35                      rotate_right(x->parent);
36                      w = x->parent->left;
37                  }
38                  if (w->right->color == Node::Color::Black
39                      && w->left->color == Node::Color::Black) {
40                      w->color = Node::Color::Red;
41                      x = x->parent;
42                  }
43                  else {
44                      if (w->left->color == Node::Color::Black) {
45                          w->right->color = Node::Color::Black;
46                          w->color = Node::Color::Red;
47                          rotate_left(w);
48                          w = x->parent->left;
49                      }
50                      w->color = x->parent->color;
51                      x->parent->color = Node::Color::Black;
52                      w->left->color = Node::Color::Black;
53                      rotate_right(x->parent);
54                      x = root;
55                  }
56              }
57          }
58          x->color = Node::Color::Black;
59      }
```

Nuevamente se trata de un código dual. En el *if* más exterior se distinguen los casos de borrar un hijo derecho o izquierdo. En ambas ramas se encuentra el mismo código intercambiando *left* por *right*. Por tanto basta analizar la primera de ellas.

Se distinguen cuatro casos:

- El hermano `w` es rojo. En ese caso forzosamente los hijos de `w` deben ser negros. Por tanto se puede intercambiar los colores del hermano y del padre y hacer una rotación a la izquierda sin violar nuevas reglas. De esta forma el nuevo hermano será forzosamente negro, por lo que este caso se transforma en alguno de los posteriores.
- El hermano `w` es negro y los dos hijos de `w` son negros. En ese caso cambiamos el color del hermano a rojo. De esta forma se equilibra el número de negros por cada rama, pero puede generar una violación de reglas en el nodo padre, que se tratará en la siguiente iteración del bucle.
- El hermano `w` es negro y el hijo izquierdo de `w` es rojo. En ese caso intercambiamos los colores de `w` y su hijo izquierdo y hacemos una rotación a la derecha. De esta forma hemos reducido este caso al siguiente.
- El hermano `w` es negro y el hijo derecho de `w` es rojo. En este caso cambiando colores en los nodos que muestra la figura 21.6.d y rotando a la izquierda se obtiene un árbol correcto que compensa el número de negros en cada rama.

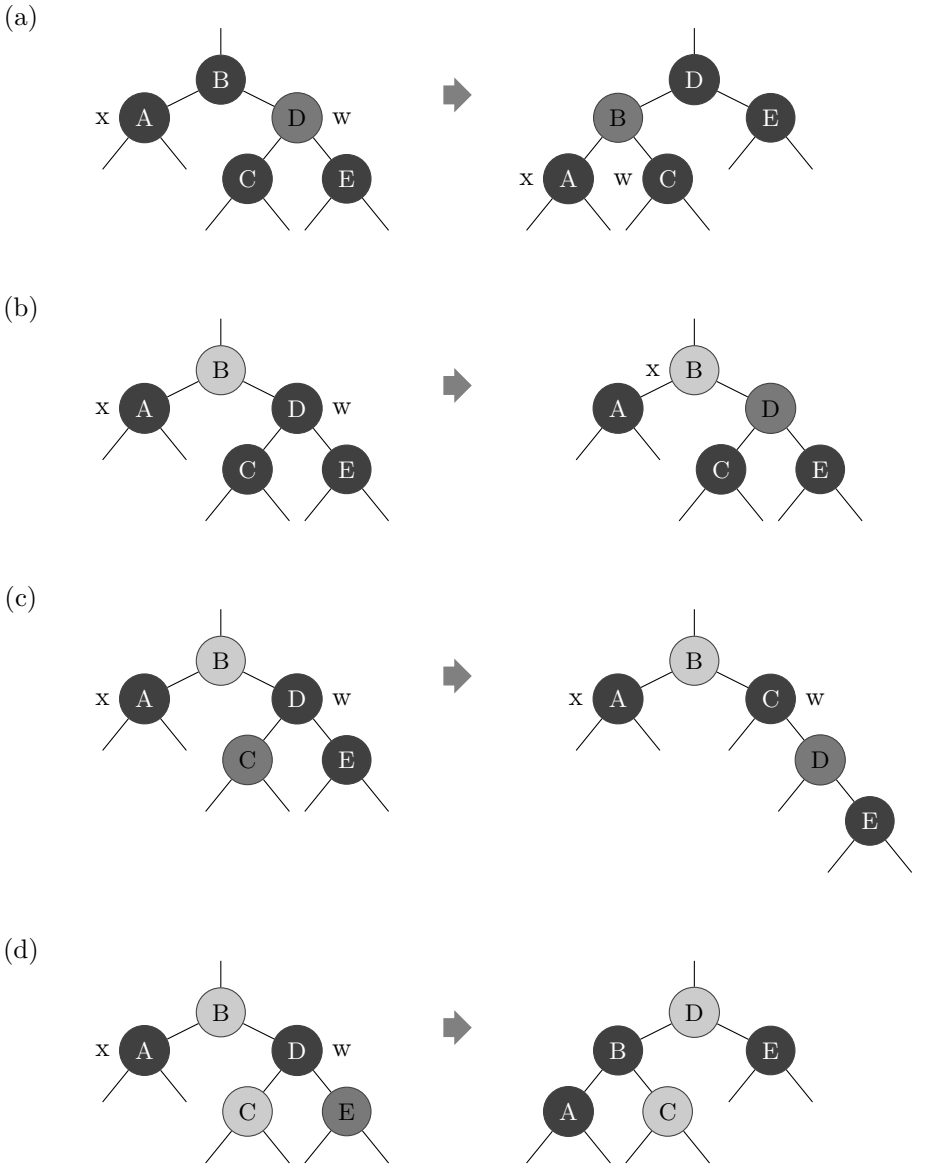


Figura 21.6: Casos contemplados en la función `remove_fixup()` según [20].

AVL trees

Los árboles AVL (Adelson-Velskii and Landis) son otra forma de árbol balanceado en el que se utiliza la altura del árbol como criterio de balanceo. Solo puede haber una diferencia de 1 entre la altura de dos ramas. Es por tanto un criterio más estricto que los *red-black trees*, lo que lo hace menos eficiente en las inserciones y borrados pero más eficiente en las lecturas.

Árboles balanceados

Los *red-black trees* son más eficientes en `insert()` y `remove()`, pero los AVL *trees* son más eficientes en `search()`.

Cada nodo tiene información adicional con la altura del árbol en ese punto. En realidad tan solo es necesario almacenar el *factor de equilibrio* que es simplemente la diferencia entre las alturas del subárbol izquierdo y el derecho. La ventaja de esta última alternativa es que es un número mucho más reducido (siempre comprendido en el rango -2 a +2) por lo que puede almacenarse en solo 3 bits.

Para insertar elementos en un árbol AVL se utiliza un procedimiento similar a cualquier inserción en árboles de búsqueda binaria, con dos diferencias:

- La inserción debe computar el factor de equilibrio en los nodos afectados.
- Finalmente hay que equilibrar el árbol si es necesario.

El equilibrado se realiza con rotaciones siguiendo el procedimiento representado en la figura 21.7. Es importante destacar que las propias funciones de rotación alteran los factores de equilibrio de los nodos involucrados (nodos `x` e `y` en la figura 21.4).

Las operaciones `insert()`, `rotate_right()`, `rotate_left()` y `remove()` sobre árboles AVL deben recalcular el factor de equilibrio en los nodos afectados. Además, en caso de dejar un árbol desequilibrado, las operaciones `insert()` y `remove()` deben equilibrar el árbol según el procedimiento descrito en la figura 21.7.

Radix tree

Aún hay otro tipo de árboles binarios que vale la pena comentar por sus implicaciones con los videojuegos. Se trata de los *árboles de prefijos*, frecuentemente llamados *tries*[1].

La figura 21.8 muestra un ejemplo de árbol de prefijos con un conjunto de enteros binarios. El árbol los representa en orden lexicográfico. Para cada secuencia binaria si empieza por 0 está en el subárbol izquierdo y si empieza por 1 en el subárbol derecho. Conforme se recorren las ramas del árbol se obtiene la secuencia de bits del número a buscar. Es decir, el tramo entre el nodo raíz y cualquier nodo intermedio define el prefijo por el que empieza el número a buscar. Por eso a este árbol se le llama *prefix tree* o *radix tree*.

Un árbol de prefijos (pero no binario) se utiliza frecuentemente en los diccionarios predictivos de los teléfonos móviles. Cada subárbol corresponde a una nueva letra de la palabra a buscar.

También se puede utilizar un árbol de prefijos para indexar puntos en un segmento de longitud arbitraria. Todos los puntos en la mitad derecha del segmento están en el subárbol derecho, mientras que todos los puntos de la mitad izquierda están en el subárbol izquierdo. Cada subárbol tiene las mismas propiedades con respecto al subsegmento que representa. Es decir, el subárbol derecho es un árbol de prefijos que representa a medio segmento derecho, y así sucesivamente. El número de niveles del árbol es ajustable dependiendo de la precisión que requerimos en el posicionamiento de los puntos.

[1] El nombre en singular es *trie*, que deriva de re**trie**ve. Por tanto la pronunciación correcta se asemeja a la de *tree*, aunque muchos autores la pronuncian como *try*.

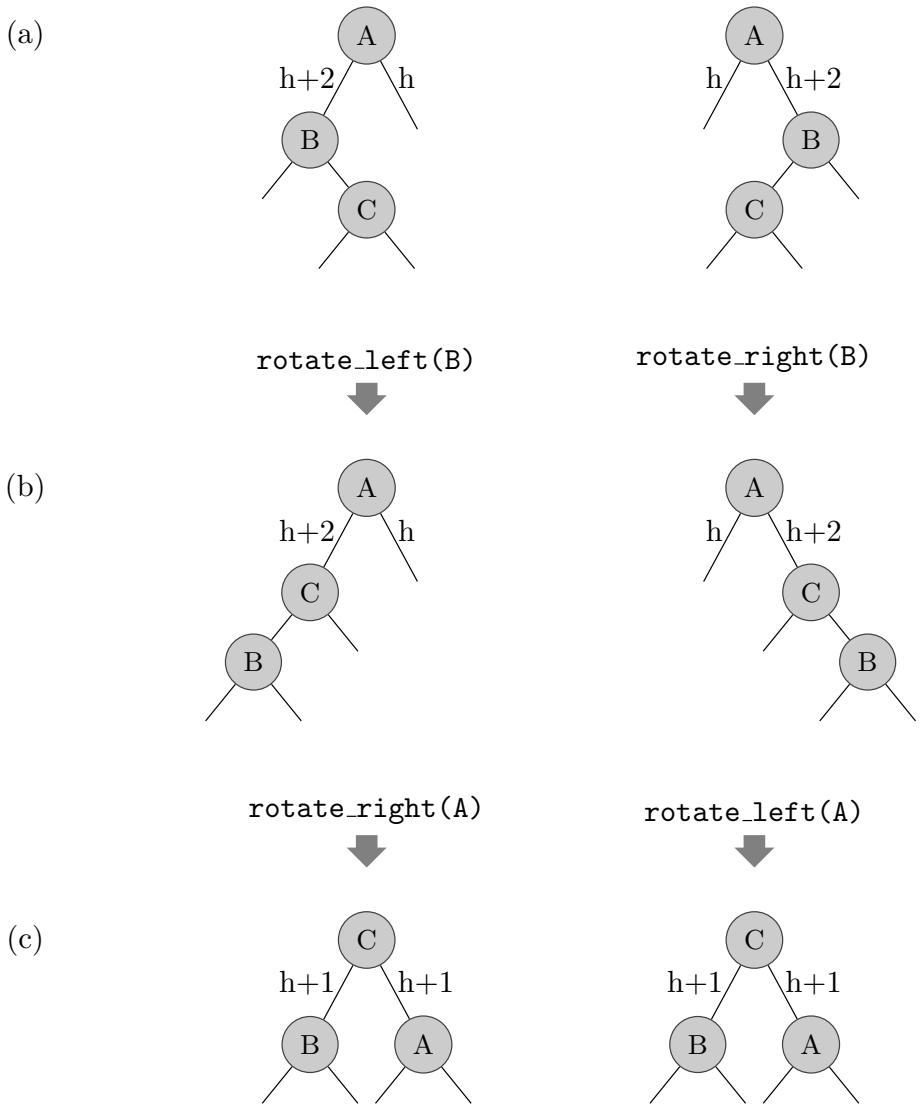


Figura 21.7: Casos contemplados en la función de equilibrado de árboles AVL.

En los árboles de prefijos la posición de los nodos está prefijada a priori por el valor de la clave. Estos árboles no realizan ninguna función de equilibrado por lo que su implementación es trivial. Sin embargo estarán razonablemente equilibrados si los nodos presentes están uniformemente repartidos por el espacio de claves.

21.1.2. Recorrido de árboles

En multitud de ocasiones es necesario recorrer los elementos de un árbol en un orden determinado. Son frecuentes los recorridos *en orden*, *en preorden*, y *en postorden*.

El recorrido *en orden* sigue el orden del campo clave. Es decir, para cualquier nodo primero se visitan los nodos del subárbol izquierdo, luego el nodo y finalmente los nodos del subárbol derecho.

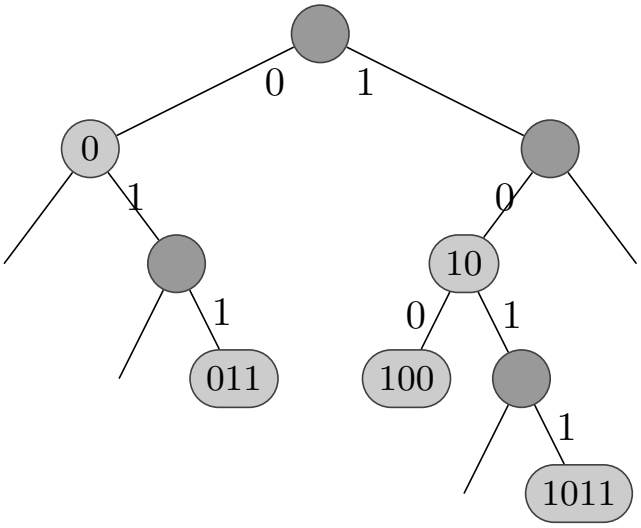


Figura 21.8: Un ejemplo de *trie* extraido de [20]. Contiene los elementos 1011, 10, 011, 100 y 0.

Listado 21.19: Recorrido *en orden* en un árbol de búsqueda binaria

```
1     template <typename Func>
2     void inorder_tree_walk(Func f) {
3         inorder_tree_walk(root, f);
4     }
5
6     template <typename Func>
7     void inorder_tree_walk(NodeType* x, Func f) {
8         if (x == 0) return;
9         inorder_tree_walk(x->left, f);
10        f(x);
11        inorder_tree_walk(x->right, f);
12    }
```

El recorrido *en preorden* visita el nodo antes de cualquiera de sus subárboles.

Listado 21.20: Recorrido *en preorden* en un árbol de búsqueda binaria

```
1     template <typename Func>
2     void preorder_tree_walk(Func f) {
3         preorder_tree_walk(root, f);
4     }
5
6     template <typename Func>
7     void preorder_tree_walk(NodeType* x, Func f) {
8         if (x == 0) return;
9         f(x);
10        preorder_tree_walk(x->left, f);
11        preorder_tree_walk(x->right, f);
12    }
```

Finalmente el recorrido *en postorden* visita el nodo después de visitar ambos subárboles.

Listado 21.21: Recorrido *en postorden* en un árbol de búsqueda binaria

```
1     template <typename Func>
2     void postorder_tree_walk(Func f) {
```

```
3         postorder_tree_walk(root, f);
4     }
5
6     template <typename Func>
7     void postorder_tree_walk(NodeType* x, Func f) {
8         if (x == 0) return;
9         postorder_tree_walk(x->left, f);
10        postorder_tree_walk(x->right, f);
11        f(x);
12    }
```

Pero para el recorrido de estructuras de datos con frecuencia es mucho mejor emplear el patrón iterador. En ese caso puede reutilizarse cualquier algoritmo de la STL.

Incluir el orden de recorrido en el iterador implica almacenar el estado necesario. Las funciones de recorrido anteriormente descritas son recursivas, por lo que el estado se almacenaba en los sucesivos marcos de pila correspondientes a cada llamada anidada. Por tanto necesitamos un contenedor con la ruta completa desde la raíz hasta el nodo actual. También tendremos que almacenar el estado de recorrido de dicho nodo, puesto que el mismo nodo es visitado en tres ocasiones, una para el subárbol izquierdo, otra para el propio nodo, y otra para el subárbol derecho.

Listado 21.22: Iterador *en orden* en un árbol de búsqueda binaria

```
1  class inorder_iterator : public std::iterator<std::
       input_iterator_tag,
2                                                 Node<KeyType>,
3                                                 ptrdiff_t,
4                                                 const Node<KeyType>*,
5                                                 const Node<KeyType>&>
                                                      {
6      typedef Node<KeyType> NodeType;
7      enum IteratorState { VisitingLeft, VisitingNode, VisitingRight
           };
8      std::vector<std::pair<NodeType*,IteratorState> > _current;
9
10 public:
11     inorder_iterator(NodeType* x) {
12         _current.push_back(std::make_pair(x,VisitingLeft));
13         goToNextNode();
14     }
15
16     const NodeType& operator*() const {
17         return *_current.back().first;
18     }
19
20     const NodeType* operator->() const {
21         return _current.back().first;
22     }
23
24     bool equal(inorder_iterator<KeyType> const& rhs) const {
25         return *this == rhs;
26     }
27
28     inorder_iterator<KeyType>& operator++() {
29         goToNextNode();
30     }
31
32     inorder_iterator<KeyType> operator++(int) {
33         inorder_iterator<KeyType> ret(*this);
34         goToNextNode();
35         return ret;
36     }
37
38 private:
39     void goToNextNode();
40 };
```

```
41
42 template<typename KeyType>
43 inline bool operator== (inorder_iterator<KeyType> const& lhs,
44                         inorder_iterator<KeyType> const& rhs) {
45     return lhs.equal(rhs);
46 }
```

En el caso del iterador *en orden* la función de recorrido sería similar a la siguiente:

Listado 21.23: Función para obtener el siguiente nodo en un iterador *en orden*.

```
1  void
2  inorder_iterator<KeyType>::goToNextNode()
3  {
4      if (_current.empty()) return;
5
6      std::pair<NodeType*,IteratorState>& last = _current.back();
7
8      if (last.second == VisitingLeft) {
9          NodeType* l = last.first->left;
10         if (l == 0) last.second = VisitingNode;
11         else {
12             _current.push_back(std::make_pair(l,VisitingLeft));
13             goToNextNode();
14         }
15     }
16     else if (last.second == VisitingNode) {
17         NodeType* r = last.first->right;
18         if (r == 0) _current.pop_back();
19         else {
20             last.second = VisitingRight;
21             _current.push_back(std::make_pair(r,VisitingLeft));
22         }
23         goToNextNode();
24     }
25     else if (last.second == VisitingRight) {
26         _current.pop_back();
27         goToNextNode();
28     }
29 }
```

Se propone como ejercicio la definición de iteradores para el recorrido en preorden y postorden.

21.1.3. *Quadtree* y *octree*

Los árboles binarios particionan de forma eficiente un espacio de claves de una sola dimensión. Pero con pequeñas extensiones es posible particionar espacios de dos y tres dimensiones. Es decir, pueden ser usados para indexar el espacio de forma eficiente.

Los *quadtrees* y los *octrees* son la extensión natural de los árboles binarios de prefijos (*tries*) para dos y tres dimensiones respectivamente. Un *quadtree* es un árbol cuyos nodos tienen cuatro subárboles correspondientes a los cuatro cuadrantes de un espacio bidimensional. Los nodos de los *octrees* tienen ocho subárboles correspondientes a los ocho octantes de un espacio tridimensional.

La implementación y el funcionamiento es análogo al de un árbol prefijo utilizado para indexar los puntos de un segmento. Adicionalmente, también se emplean para indexar segmentos y polígonos.

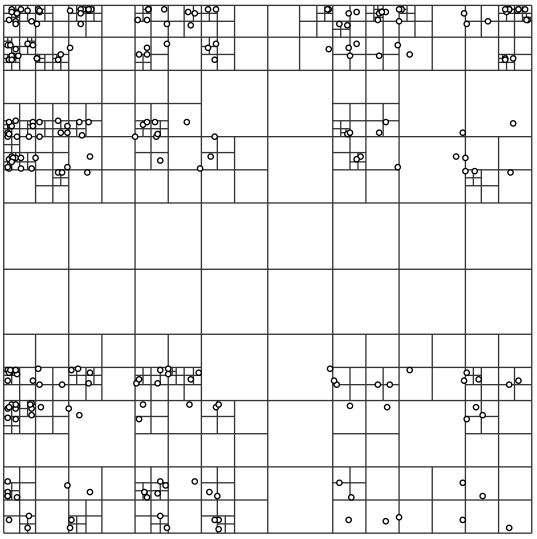


Figura 21.9: Ejemplo de *quadtree* de puntos. Fuente: Wikipedia.

Cuando se utilizan para indexar segmentos o polígonos puede ocurrir que un mismo segmento cruce el límite de un cuadrante o un octante. En ese caso existen dos posibles soluciones:

- Hacer un recortado (*clipping*) del polígono dentro de los límites del cuadrante u octante.
- Poner el polígono en todos los cuadrantes u octantes con los que intersecta.

En este último caso es preciso disponer de alguna bandera asociada a los polígonos para no recorrerlos más veces de las necesarias.

Simon Perreault distribuye una implementación sencilla y eficiente de *octrees* en C++[2]. Simplificando un poco esta implementación los nodos son representados de esta forma:

Listado 21.24: Representación de nodos en un *octree*.

```
 1 enum NodeType { BranchNode, LeafNode };
 2
 3 class Node {
 4 public:
 5     NodeType type() const;
 6
 7 private:
 8     NodeType type_ : 2;
 9 };
10
11 class Branch : public Node {
12 public:
13     Node*& child( int x, int y, int z );
14     Node*& child( int index );
15
16 private:
```

[2]En el momento de editar estas notas se distribuye bajo la GPL en `http://nomis80.org/code/octree.html`.

```
17      Node* children[2][2][2];
18  };
19
20  class Leaf : public Node {
21  public:
22      Leaf( const T& v );
23
24      const T& value() const;
25      T& value();
26      void setValue( const T& v );
27
28  private:
29      T value_;
30  };
```

Esta representación de árboles diferencia entre nodos hoja y nodos de ramificación. Los valores solo se almacenan en los nodos hoja y éstos no tienen la sobrecarga de los punteros a los ocho subárboles. Por contra, los nodos de ramificación no tienen sobrecarga de valores asociados, puesto que para la inserción de un elemento puede ser necesario añadir un número de nodos de ramificación sin valor alguno.

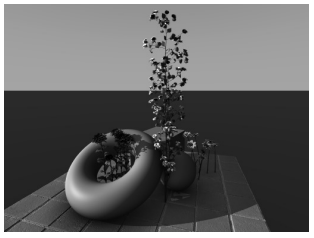


Figura 21.10: Ejemplo del motor de *Sparse Voxel Octree* de nVidia.

El uso básico es muy sencillo. El contenido a incluir puede ser cualquier cosa, desde simples valores (color de un punto), pasando por un *voxel* (pixel 3D) hasta polígonos o poliedros. El tipo de contenido puede ser también una referencia a un objeto gráfico. De esta forma se podría incluir el mismo elemento (e.g. un polígono) en múltiples nodos hoja. En el capítulo siguiente veremos cómo la técnica de referencias con contador puede ayudar en casos como éste.

Listado 21.25: Representación de nodos en un *octree*.

```
1  #include "octree.h"
2
3  int main()
4  {
5      Octree<double> o(4096);
6      o(1,2,3) = 3.1416;
7      o.erase(1,2,3);
8  }
```

La línea [5] construye un *octree* de 4096 puntos de ancho en cada dimensión. Esta implementación requiere que sea una potencia de dos, siendo posible indexar $4096 \times 4096 \times 4096$ nodos.

La regularidad de los *octree* los hacen especialmente indicados para la paralelización con GPU y recientemente están teniendo cierto resurgimiento con su utilización en el renderizado de escenas con *raycasting* o incluso *raytracing* en una técnica denominada *Sparse Voxel Octree*.

La propia nVidia[3] ha desarrollado una biblioteca que implementa esta técnica de *sparse voxel octree* sobre GPU con CUDA (Compute Unified Device Architecture), y se distribuye bajo licencia Apache. Otra excelente fuente de información es la tesis de Cyril Crassin de INRIA[4] que explica los fundamentos teóricos de GigaVoxels, también basada en *octrees*.

[3] Ver `http://research.nvidia.com/publication/efficient-sparse-voxel-octrees`

[4] Disponible en línea en `http://maverick.inria.fr/Membres/Cyril.Crassin/thesis/CCrassinThesis_EN_Web.pdf`

21.2. Patrones de diseño avanzados

En el módulo 1 ya se expusieron un buen número de patrones. En esta sección completaremos la colección con algunos patrones muy utilizados en todo tipo de aplicaciones.

21.2.1. Smart pointers

Los punteros inteligentes (*smart pointers*) son tipos de datos que simplifican de alguna manera la gestión de la memoria dinámica. Facilitan la gestión del ciclo de vida de las variables dinámicas para evitar los problemas frecuentemente asociados a los punteros, especialmente la liberación de la memoria.

La biblioteca estándar de C++ incorpora una plantilla denominada `auto_ptr`. Su objetivo es envolver un puntero normal de tal forma que la destrucción del puntero lleve consigo también la destrucción del objeto apuntado. Por lo demás, un `auto_ptr` se comporta como si se tratara del propio puntero. Por ejemplo, es frecuente encontrar código como el que sigue:

Listado 21.26: Ejemplo de uso inseguro de punteros.

```
1     T* p = new T();
2
3     // cuerpo de la función
4
5     delete p;
```

Este fragmento tiene dos problemas:

- Es relativamente fácil olvidar llamar a `delete`. Conforme evoluciona el código pueden aparecer puntos de retorno que no invocan al destructor.
- En esta secuencia no es posible garantizar que el flujo del programa será secuencial. Es perfectamente posible que en medio del código de la función se eleve una excepción. En ese caso no se ejecutará el `delete`. Por supuesto siempre es posible utilizar construcciones `try`/`catch` pero el código cada vez se haría menos legible.

Bjarne Stroustrup inventó una técnica de aplicación general para resolver este tipo de problemas. Se llama RAII (Resource Acquisition Is Initialization) y básicamente consiste en encapsular las operaciones de adquisición de recursos y liberación de recursos en el constructor y destructor de una clase normal. Esto es precisamente lo que hace `auto_ptr` con respecto a la reserva de memoria dinámica. El mismo código del fragmento anterior puede reescribirse de forma segura así:

Listado 21.27: Ejemplo de uso seguro de punteros.

```
1     auto_ptr<T> p = new T();
2
3     // cuerpo de la función
```

No importa el camino que siga el programa, aunque se eleve una excepción. En el momento en que se abandone el bloque en el que se ha declarado el `auto_ptr` se invocará a su destructor, que a su vez invocará `delete`.

Como puede verse hemos ligado el tiempo de vida del objeto construido en memoria dinámica al tiempo de vida del `auto_ptr`, que suele ser una variable automática o un miembro de clase. Se dice que el `auto_ptr` posee al objeto dinámico. Pero puede ceder su posesión simplemente con una asignación o una copia a otro `auto_ptr`.

Listado 21.28: Cesión de la posesión del objeto dinámico.

```
1     auto_ptr<T> q(p);
2     auto_ptr<T> r;
3     p->f(); // error (NULL ref)
4     q->f(); // ok
5     r = q;
6     q->f(); // error (NULL ref)
7     r->f(); // ok
```

Es decir, `auto_ptr` garantiza que solo hay un objeto que posee el objeto dinámico. También permite desligar el objeto dinámico del `auto_ptr` para volver a gestionar la memoria de forma explícita.

Listado 21.29: Recuperación de la propiedad del objeto dinámico.

```
1     T* s = r.release();
2     delete s;
```

Nunca se deben usar `auto_ptr` en contenedores estándar, porque los contenedores de la STL asumen una semántica de copia incompatible con la del `auto_ptr`. La copia de un `auto_ptr` no genera dos objetos equivalentes.

Esta limitación, que no es detectada en tiempo de compilación, es una de las motivaciones de un completo rediseño de esta funcionalidad para el estándar C++ de 2011. Aún sigue soportando `auto_ptr` pero se desaconseja su uso en favor de `unique_ptr`. El nombre deriva de que, al igual que `auto_ptr`, garantiza que solo un `unique_ptr` puede estar apuntando a un mismo recurso. Sin embargo, a diferencia de `auto_ptr` no es posible copiarlos. Sin embargo existe la posibilidad de transferencia de propiedad entre `unique_ptr` utilizando la nueva semántica de movimiento del estándar C++11.

Listado 21.30: Ejemplo de uso de `unique_ptr`.

```
1     unique_ptr<T> p(new T());
2     unique_ptr<T> q;
3     q = p; // error (no copiable)
4     q = std:move(p);
```

La plantilla `unique_ptr` no tiene un constructor de copia, pero sí cuenta con un constructor de movimiento. Este nuevo constructor se aplica cuando el parámetro es un *rvalue*, es decir, una expresión del tipo de las que aparecen en el lado derecho de una asignación (de ahí el nombre, *right value*) o un valor de retorno de función, o la copia temporal de un parámetro pasado por copia (ahora se puede pasar también por movimiento). Este tipo de expresiones se caracterizan en C++ porque generan un *temporary*, una variable temporal.

La semántica de movimiento resuelve el problema de la generación inconsciente de multitud de variables temporales y la separación entre constructor de copia y constructor de movimiento permite detectar en tiempo de compilación los problemas semánticos. La copia siempre debería generar dos objetos equivalentes.

Tanto `auto_ptr` como `unique_ptr` proporcionan un método sencillo para gestionar variables en memoria dinámica casi como si se tratara de variables automáticas. Por ejemplo:

Listado 21.31: Función que reserva memoria dinámica y traspasa la propiedad al llamador. También funcionaría correctamente con `auto_ptr`.

```
1 unique_ptr<T> f() {
2     unique_ptr<T> p(new T());
3     // ...
4     return p;
5 }
```

La función `f()` devuelve memoria dinámica. Con simples punteros eso implicaba que el llamante se hacía cargo de su destrucción, de controlar su ciclo de vida. Con esta construcción ya no es necesario. Si el llamante ignora el valor de retorno éste se libera automáticamente al destruir la variable temporal correspondiente al valor de retorno. Si en cambio el llamante asigna el valor de retorno a otra variable `unique_ptr` entonces está asumiendo la propiedad y se liberará automáticamente cuando el nuevo `unique_ptr` sea destruido.

Las nuevas características de la biblioteca estándar para la gestión del ciclo de vida de la memoria dinámica están ya disponibles en los compiladores libres GCC y clang. Tan solo hay que utilizar la opción de compilación `-stdc++0x`.

Tanto con `auto_ptr` como con `unique_ptr` se persigue que la gestión de memoria dinámica sea análoga a la de las variables automáticas con semántica de copia. Sin embargo no aprovechan la posibilidad de que el mismo contenido de memoria sea utilizado desde varias variables. Es decir, para que la semántica de copia sea la natural en los punteros, que se generen dos objetos equivalentes, pero sin copiar la memoria dinámica. Para ese caso el único soporte que ofrecía C++ hasta ahora eran los punteros y las referencias. Y ya sabemos que ese es un terreno pantanoso.

La biblioteca estándar de C++11 incorpora dos nuevas plantillas para la gestión del ciclo de vida de la memoria dinámica que ya existían en la biblioteca Boost: `shared_ptr` y `weak_ptr`. Ambos cooperan para disponer de una gestión de memoria muy flexible. La plantilla `shared_ptr` implementa una técnica conocida como *conteo de referencias*.

Cuando se asigna un puntero por primera vez a un `shared_ptr` se inicializa un contador interno a 1. Este contador se almacena en memoria dinámica y es compartido por todos los `shared_ptr` que apunten al mismo objeto. Cuando se asigna este `shared_ptr` a otro `shared_ptr` o se utiliza el constructor de copia, se incrementa el contador interno. Cuando se destruye un `shared_ptr` se decrementa el contador interno. Y finalmente cuando el contador interno llega a 0, se destruye automáticamente el objeto dinámico.

Listado 21.32: Ejemplos de uso de `shared_ptr`.

```
1     shared_ptr<T> p(new T());
2     shared_ptr<T> q;
3     {
4         q = p;
5         shared_ptr<T> r(p);
6         // ...
```

```
7       }
8       // ...
```

En la línea [1] se construye un `shared_ptr` que apunta a un objeto dinámico. Esto pone el contador interno de referencias a 1. En la línea [4] se asigna este `shared_ptr` a otro. No se copia el objeto dinámico, sino solo su dirección y la del contador de referencias, que además es automáticamente incrementado (pasa a valer 2). En la línea [5] se utiliza el constructor de copia de otro `shared_ptr`, que nuevamente copia solo el puntero y el puntero al contador de referencias, además de incrementar su valor (pasa a valer 3). En la línea [7] se destruye automáticamente `r`, con lo que se decrementa el contador de referencias (vuelve a valer 2). Cuando acabe el bloque en el que se han declarado `p` y `q` se destruirán ambas variables, y con ello se decrementará dos veces el contador de referencias. Al llegar a 0 automáticamente se invocará el operador `delete` sobre el objeto dinámico.

El conteo de referencias proporciona una poderosa herramienta para simplificar la programación de aplicaciones con objetos dinámicos. Los `shared_ptr` pueden copiarse o asignarse con total libertad y con una semántica intuitiva. Pueden emplearse en contenedores de la STL y pasarlos por valor libremente como parámetros a función o como valor de retorno de una función. Sin embargo no están totalmente exentos de problemas. Considera el caso en el que `main()` tiene un `shared_ptr` apuntando a una clase `A` y ésta a su vez contiene directa o indirectamente un `shared_ptr` que vuelve a apuntar a `A`. Tendríamos un ciclo de referencias y el contador de referencias con un balor de 2. En caso de que se destruyera el `shared_ptr` inicial seguiríamos teniendo una referencia a `A`, por lo que no se destruirá.

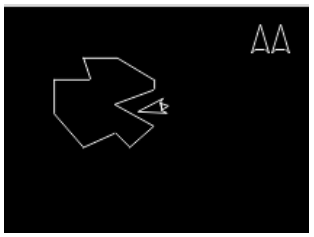


Figura 21.11: El propio creador de C++ pone como ejemplo el videojuego *asteroids* para explicar las extensiones a la biblioteca estándar.

Para romper los ciclos de referencias la biblioteca estándar incluye la plantilla `weak_ptr`. Un `weak_ptr` es otro *smart pointer* a un objeto que se utiliza en estas condiciones:

1. Solo se necesita acceso al objeto si existe.
2. Puede ser borrado por otros en cualquier momento.
3. Debe ser destruido tras su último uso.

Bjarne Stroustrup [5] pone un ejemplo que tiene mucho que ver con la programación de videojuegos. Consideremos el caso del juego de los asteroides. Todos los asteroides son poseídos por "el juego" pero cada asteroide tiene que seguir los movimientos de los asteroides vecinos para detectar colisiones. Una colisión lleva a la destrucción de uno o más asteroides. Cada asteroide debe almacenar una lista de los asteroides vecinos. Pero el hecho de pertenecer a esa lista no mantiene al asteroide vivo. Por tanto el uso de `shared_ptr` sería inapropiado. Por otro lado un asteroide no debe ser destruido mientras otro asteroide lo examina (para calcular los efectos de una colisión, por ejemplo), pero debe llamarse al destructor en algún momento para liberar los recursos asociados. Necesitamos una lista de asteroides que *podrían* estar vivos y una forma de sujetarlos por un tiempo. Eso es justo lo que hace `weak_ptr`.

Listado 21.33: Esquema de funcionamiento del propietario de los asteroides. Usa `shared_ptr` para representar propiedad.

```
1     vector<shared_ptr<Asteroid>> va(100);
2     for (int i=0; i<va.size(); ++i) {
3         // ... calculate neighbors for new asteroid ...
4         va[i].reset(new Asteroid(weak_ptr<Asteroid>(va[neighbor])))
              ;
5         launch(i);
```

[5] http://www.research.att.com/~bs/C++0xFAQ.html#std-weak_ptr

```
6       }
7       // ...
```

El cálculo de colisiones podría tener una pinta similar a esto:

Listado 21.34: Esquema de funcionamiento de la detección de colisiones. Usa `weak_ptr` para representar la relación con los vecinos.

```
1       if (shared_ptr<Asteroid> q = p.lock()) {
2           // ... Asteroid still alive: calculate ...
3       }
4       else {
5           // ... oops: Asteroid already destroyed
6       }
```

Aunque el propietario decidiera terminar el juego y destruir todos los asteroides (destruyendo los correspondientes `shared_ptr` que representan la relación de propiedad) todo funcionaría con normalidad. Cada asteroide que se encuentra en mitad del cálculo de colisión todavía terminaría correctamente puesto que el método `lock()` proporciona un `shared_ptr` que no puede quedar invalidado.

Por último merece la pena comentar en esta sección un conjunto de reglas para escribir código correcto con *smart pointers*:

- Siempre que aparezca un operador `new` debe ser en un constructor de un *smart pointer*.
- Evitar el uso de *smart pointers* sin nombre (e.g. *temporaries*).

La primera regla impide tener punteros normales coexistiendo con los *smart pointers*. Eso solo puede generar quebraderos de cabeza, puesto que el *smart pointer* no es capaz de trazar los accesos al objeto desde los punteros normales.

La segunda regla garantiza la liberación correcta de la memoria en presencia de excepciones[6]. Veamos un ejemplo extraído de la documentación de Boost:

Listado 21.35: Uso de *smart pointers* en presencia de excepciones.

```
1  void f(shared_ptr<int>, int);
2  int g();
3
4  void ok() {
5      shared_ptr<int> p(new int(2));
6      f(p, g());
7  }
8
9  void bad() {
10     f(shared_ptr<int>(new int(2)), g());
11 }
```

Para entender por qué la linea 10 es peligrosa basta saber que el orden de evaluación de los argumentos no está especificado. Podría evaluarse primero el operador `new`, después llamarse a la función `g()`, y finalmente no llamarse nunca al constructor de `shared_ptr` porque `g()` eleva una excepción.

[6] Este caso ha sido descrito en detalle por Herb Sutter en `http://www.gotw.ca/gotw/056.htm`.

En la mayoría de las bibliotecas de relativa complejidad encontramos algún tipo de *smart pointer*. En Ogre ya hemos visto `Ogre::SharedPtr`, en ZeroC Ice hemos visto `IceUtil::Handle`, en Boost hay una amplia colección de *smart pointers* que incluye `boost::shared_ptr` y `boost::unique_ptr`. Ahora que el nuevo estándar C++ incluye conteo de referencias veremos una progresiva evolución de las bibliotecas para adoptar la versión estándar. Mientras tanto, es muy importante utilizar en cada biblioteca los mecanismos que incluye y no mezclarlos con otras bibliotecas.

Un problema evidente al usar `shared_ptr` es la existencia de dos objetos en el *heap* por cada objeto. Si construimos un objeto con `new` y entonces lo asignamos a un `shared_ptr` está claro que el propio `shared_ptr` deberá construir en el *heap* un entero con el que lleva el conteo de referencias y que será compartido por todos los `shared_ptr` que apunten al mismo objeto. Esto es bastante grave en la práctica porque cada objeto reservado en el *heap* lleva algo de información asociada para gestionar las zonas libres y ocupadas. Si se reservan multitud de pequeños objetos dinámicos estaríamos dilapidando memoria.

En C++11 se evita el problema recomendando encarecidamente que no se utilice `new` ni `delete` en absoluto. En su lugar se proporciona un objeto función denominado `make_shared` y se delega la llamada a `delete` a los smart pointers.

Pimpl con `auto_ptr` o `unique_ptr`

Ya hemos visto en el módulo 1 el patrón idiomático del *handle-body* o *Pimpl*. Sin embargo con el uso de *smart pointers* puede conseguirse una implementación más elegante del patrón.

Un buen compromiso entre automatización de la gestión de memoria y flexibilidad en la implementación de este patrón es la plantilla `auto_ptr` (o `unique_ptr` para C++11) de la biblioteca estándar de C++. La implementación del patrón *Pimpl* puede simplificarse aún más como recomienda Herb Sutter[7]:

Listado 21.36: Ejemplo mejorado del patrón Pimpl (archivo de cabecera).

```
1 class C {
2 public:
3     C();
4     /*...*/
5 private:
6     class CImpl;
7     auto_ptr<CImpl> pimpl_;
8 };
```

La diferencia clave es la declaración del puntero a la implementación como un `auto_ptr` en la línea (7). La declaración anticipada de la clase implementación se ha metido también en la parte privada del *handle* para mejorar la ocultación.

Listado 21.37: Ejemplo mejorado del patrón Pimpl (archivo de implementación).

```
1 class C::CImpl { /*...*/ };
2
3 C::C() : pimpl_( new CImpl ) { }
```

[7] Por ejemplo, en
`http://www.gotw.ca/publications/using_auto_ptr_effectively.htm`.

Ahora no es necesario incluir un destructor explícitamente porque el destructor por defecto llamará a los destructores de los miembros, en particular de `pimpl_`. Y el destructor de un `auto_ptr` llama automáticamente al operador `delete` con el puntero interno.

21.2.2. Command

El patrón *command* (se traduciría como *orden* en castellano) se utiliza frecuentemente en interfaces gráficas para el manejo de las órdenes del usuario. Consiste en encapsular las peticiones en objetos que permiten desacoplar la emisión de la orden de la recepción, tanto desde el punto de vista lógico como temporal.

Problema

Existe un gran número de situaciones en las que la sincronía inherente a la invocación directa a métodos resulta poco conveniente:

- La invocación directa solamente involucra a emisor y receptor de la orden, por lo que resulta complicado trazar la actividad del sistema en otros componentes (barras de progreso, capacidad de deshacer las órdenes ejecutadas, ayuda contextual, etc.).
- En algunas ocasiones es necesario un modelo de ejecución transaccional, o con limitaciones de orden. Así, por ejemplo si se ejecuta una acción también deben ejecutarse todas las acciones relacionadas. Y si no se ejecuta una acción deben deshacerse todas las relacionadas. Las acciones sobre un mundo virtual (e.g. un MMORPG) deben garantizar la ejecución en orden causal para todos los jugadores (la causa precede al efecto).
- En ocasiones conviene grabar y reproducir una secuencia de órdenes (e.g para la implementación de macros o simplemente para la prueba del juego).
- Muchas acciones conllevan la interacción con el usuario en forma de *wizards* o cuadros de diálogo para configurar la acción. El patrón *command* permite que el objeto orden sea creado en el momento de mostrar el *wizard*, que el usuario configure el objeto mediante la interacción con el *wizard*, y finalmente, al cerrar el *wizard* se desencadena el proceso de emisión del mensaje. De esta forma la orden no necesita nada de código de interfaz de usuario.
- La mayoría de los juegos actuales son programas multi-hilo. Las órdenes pueden ser generadas desde multitud de hilos, y el procesamiento de éstas puede corresponder a otro conjunto de hilos diferente. El patrón *command* proporciona un método sencillo para desacoplar productores y consumidores de órdenes.
- En los juegos en red necesitamos ejecutar órdenes en todos los ordenadores participantes. El patrón *command* facilita la serialización de las órdenes sin más que serializar los objetos que las representan.
- Muchos juegos añaden algún tipo de consola para interactuar directamente con el motor empleando un intérprete de órdenes. El patrón *command* permite sintetizar órdenes en el juego como si se hubieran producido en el propio juego, lo que facilita enormemente la prueba y depuración.

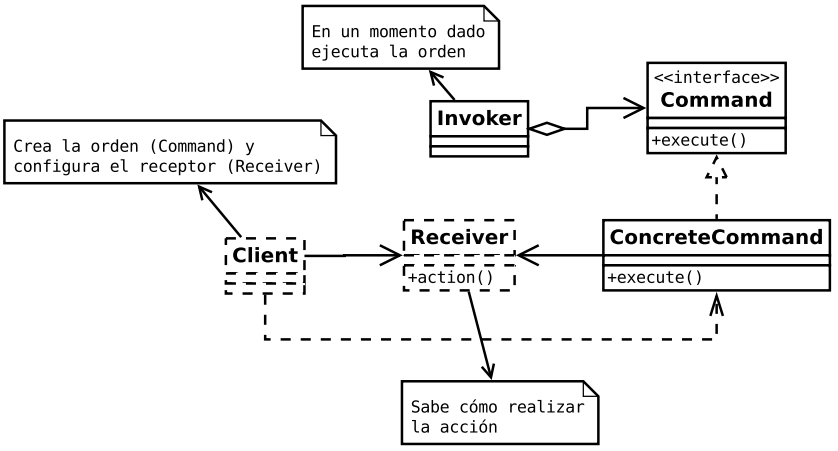


Figura 21.12: Estructura del patrón *command*.

Solución

Figura 21.13: Las acciones de los personajes de un juego son perfectas para el patrón *command*.

La figura 21.12 muestra un diagrama de clases con las entidades involucradas. El cliente es el que crea los objeto *command* concretos y los asocia con el receptor de la acción. Posteriormente, y de forma totalmente desacoplada, un invocador llamará al método `execute()` de cada objeto orden creado. Los objetos *command* concretos implementan el método `execute()`, normalmente delegando total o parcialmente sobre el receptor de la acción.

Un ejemplo de aplicación en un videojuego podría ser el que se muestra en la figura 21.14.

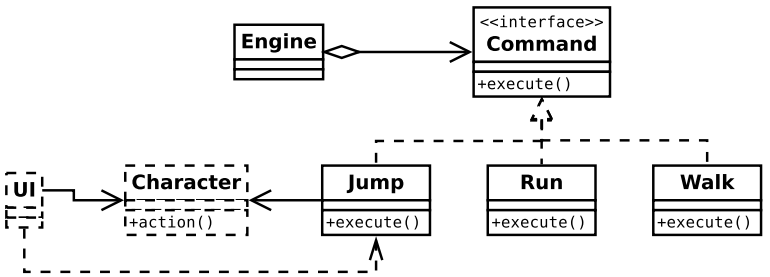


Figura 21.14: Ejemplo de aplicación del patrón *command*.

El interfaz de usuario crea las órdenes a realizar por el personaje o los personajes que están siendo controlados, así como la asociación con su personaje. Estas acciones se van procesando por el motor del juego, posiblemente en paralelo.

Implementación

En términos generales el patrón *command* permite descargar más o menos inteligencia sobre el objeto `ConcreteCommand`. Se juega entre los dos posibles extremos.

- El objeto `ConcreteCommand` no realiza ninguna función por sí mismo, sino que delega todas las acciones en el objeto `Receiver`. A este tipo de órdenes se les llama *forwarding commands*.
- El objeto `ConcreteCommand` implementa absolutamente todo, sin delegar nada en absoluto al objeto `Receiver`.

Entre estos dos extremos se encuentran las órdenes que realizan algunas funciones pero delegan otras en el receptor. En general a todo este tipo de órdenes se les denomina *active commands*.

Desde el punto de vista de la implementación hay poco que podamos añadir a una orden activa. Tienen código de aplicación específico que hay que añadir en el método `execute()`.

Sin embargo, los *forwarding commands* actúan en cierta forma como si se tratara de punteros a función. El `Invoker` invoca el método `execute()` del objeto orden y éste a su vez ejecuta un método del objeto `Receiver` al que está asociado. En [9] se describe una técnica interesante para este fin, los *generalized functors* o adaptadores polimórficos para objetos función. Se trata de una plantilla que encapsula cualquier objeto, cualquier método de ese objeto, y cualquier conjunto de argumentos para dicho método. Su ejecución se traduce en la invocación del método sobre el objeto con los argumentos almacenados. Este tipo de *functors* permiten reducir sensiblemente el trabajo que implicaría una jerarquía de órdenes concretas. Boost implementa una técnica similar en la plantilla `function`, que ha sido incorporada al nuevo estándar de C++ (en la cabecera `functional`). Por ejemplo:

Listado 21.38: Ejemplo de uso de *generalized functors*.

```
#include <functional>

using namespace std;

int f1(const char* s) { return 0; }

struct f2 {
    int operator() (const char* s) { return 0; }
};

struct A {
    int fa(const char* s) { return 0; }
};

int
main()
{
    function<int (const char*)> f;

    f = f1; f("test1");
    f = f2(); f("test2");
        A a;
        auto f3 = bind1st(mem_fun(&A::fa), &a);
    f = f3; f("test3");
}
```

La plantilla `function` se instancia simplemente indicando la signatura de las llamadas que encapsula. A partir de ahí se puede asignar cualquier tipo de objeto que cumpla la signatura, incluyendo funciones normales, métodos o *functors* de la STL, *functors* implementados a mano, etc.

Consideraciones

- El patrón *command* desacopla el objeto que invoca la operación del objeto que sabe cómo se realiza.
- Al contrario que la invocación directa, las órdenes son objetos normales. Pueden ser manipulados y extendidos como cualquier otro objeto.
- Las órdenes pueden ser agregadas utilizando el patrón *composite*.
- Las órdenes pueden incluir transacciones para garantizar la consistencia sin ningún tipo de precaución adicional por parte del cliente. Es el objeto `Invoker` el que debe reintentar la ejecución de órdenes que han abortado por un interbloqueo.
- Si las órdenes a realizar consisten en invocar directamente un método o una función se puede utilizar la técnica de *generalized functors* para reducir el código necesario sin necesidad de implementar una jerarquía de órdenes.

21.2.3. Curiously recurring template pattern

Este patrón fue inicialmente descrito y bautizado por James O. Coplien en [19]. Se trata de un patrón que ya se utilizaba años antes, desde los primeros tiempos de las plantillas de C++.

Problema

El patrón CRTP (Curiously Recurring Template Pattern) pretende extraer funcionalidad común a varias clases, pero que requieren especialización parcial para cada una de ellas.

Solución

La solución pasa por una interesante recurrencia.

Listado 21.39: Estructura básica del patrón CRTP.

```
1 template<typename T> class Base;
2
3 class Derived: public Base<Derived> {
4     // ...
5 };
```

La clase derivada hereda de una plantilla instanciada para ella misma. La clase base cuenta en su implementación con un tipo que deriva de ella misma. Por tanto la propia clase base puede llamar a funciones especializadas en la clase derivada.

Implementación

Se han propuesto multitud de casos donde puede aplicarse este patrón. Nosotros destacaremos en primer lugar su uso para implementar visitantes alternativos a los ya vistos en el módulo 1.

Recordaremos brevemente la estructura del patrón visitante tal y como se contó en el módulo 1. Examinando la figura 21.15 podemos ver que:

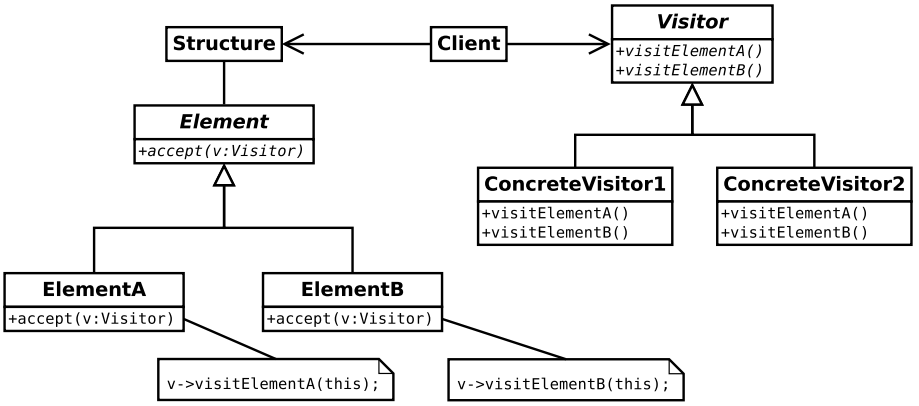


Figura 21.15: Diagrama de clases del patrón Visitor

- La clase base `Visitor` (y por tanto todas sus clases derivadas) es tremendamente dependiente de la jerarquía de objetos visitables de la izquierda. Si se implementa un nuevo tipo de elemento `ElementC` (una nueva subclase de `Element`) tendremos que añadir un nuevo método `visitElementB()` en la clase `Visitor` y con ello tendremos que reescribir todos y cada uno de las subclases de `Visitor`. Cada clase visitable tiene un método específico de visita.

- La jerarquía de elementos visitables no puede ser una estructura arbitraria, debe estar compuesta por subclases de la clase `Element` e implementar el método `accept()`.

- Si se requiere cambiar la estrategia de visita. Por ejemplo, unificar el método de visita de dos tipos de elementos, es preciso cambiar la jerarquía de objetos visitables.

- El orden de visita de los elementos agregados está marcado por la implementación concreta de las funciones `accept()` o `visitX()`. O bien se introduce el orden de recorrido en los métodos `accept()` de forma que no es fácil cambiarlo, o bien se programa a medida en los métodos `visitX()` concretos. No es fácil definir un orden de recorrido de elementos (en orden, en preorden, en postorden) común para todos las subclases de `Visitor`.

En general, se considera que el patrón *visitor* introduce un excesivo acoplamiento en el código y resulta tremendamente invasivo. Sin embargo, el patrón CRTP permite aliviar gran parte de los problemas.

La jerarquía de visitables implementa el método `accept()` exclusivamente para que puedan elegir el método `visit()` correcto de la clase derivada de `Visitor`. Por eso se le llama también despachado doble. El despachado de la función virtual `accept()` selecciona la subclase de `Element` concreta y a su vez ese elemento concreto desencadena el despachado de `visitX()` que selecciona la subclase de `Visitor` concreta. El segundo despachado es esencial para cualquier recorrido. Sin embargo el primer despachado no siempre es necesario si conocemos de alguna manera el tipo a visitar. Por ejemplo, en el ejemplo del patrón *visitor* mostrado en el módulo 1 el tipo de objetos es completamente fijo. Sabemos que hay un objeto `Scene` que contiene un número variable de objetos `ObjectScene`. Otra forma de realizar este primer despachado podría ser utilizando RTTI (Run Time Type Information) u otro mecanismo de introspección.

En este caso en que no sea necesario el primer despachado virtual se puede lograr de una manera mucho más eficiente sin ni siquiera usar funciones virtuales, gracias al patrón CRTP. Por ejemplo, el mismo ejemplo del módulo 1 quedaría así:

Listado 21.40: Visitante genérico usando el patrón CRTP.

```
1  struct ObjectScene {
2      string name;
3      Point position;
4      int weight;
5  };
6
7  struct Scene {
8      template <typename Derived> friend class Visitor;
9      string name;
10     vector<ObjectScene> objects;
11 };
12
13 template <typename Derived>
14 class Visitor {
15 public:
16     void traverseObject(ObjectScene* o) {
17         getDerived().visitObject(o);
18     }
19     void traverseScene(Scene* s) {
20         getDerived().visitScene(s);
21         for (auto o : s->objects)
22             traverseObject(o);
23     }
24     void visitObject(ObjectScene* o) {}
25     void visitScene(Scene* s) {}
26 private:
27     Derived& getDerived() {
28         return *static_cast<Derived*>(this);
29     }
30 };
31
32 class NameVisitor : public Visitor<NameVisitor> {
33     vector<string> _names;
34 public:
35     void visitObject(ObjectScene* o) {
36         _names.push_back(o->name);
37     }
38     void visitScene(Scene* s) {
39         cout << "The scene '" << s->name << "' has the following
               objects:"
40              << endl;
41         for (auto n : _names) cout << n << endl;
42     }
43 };
44
45 class BombVisitor : public Visitor<BombVisitor> {
46     Bomb _bomb;
47 public:
48     BombVisitor(const Bomb& bomb) : _bomb(bomb) {}
49     void visitObject(ObjectScene* o) {
50         Point new_pos = calculateNewPosition(o->position,
51                                              o->weight,
52                                              _bomb.intensity);
53         o->position = new_pos;
54     }
55 };
```

Como puede observarse, ahora no tocamos en absoluto la jerarquía de visitables (no se necesita método `accept`) y no hay ninguna función virtual involucrada. En el `Visitor` distinguimos entre las funciones de recorrido, que son comunes a cualquier otro `Visitor` y las de visita, que se especifican por cada visitante concreto. Su uso es prácticamente igual de sencillo:

Listado 21.41: Utilización del visitante basado en CRTP.

```
1      Scene* scene = createScene();
2      NameVisitor nv;
3      nv.traverseScene(scene);
4      // ...
5      // bomb explosion occurs
6      BombVisitor bv(bomb);
7      bv.traverseScene(scene);
```

Pero la utilidad del patrón no se limita a implementar visitantes. Es un mecanismo genérico para implementar *mixins*. En programación orientada a objetos un *mixin* es una clase que proporciona funcionalidad para ser reusada directamente por sus subclases. Es decir, las subclases no especializan al *mixin* sino que simplemente incorporan funcionalidad derivando de él.

Un ejemplo clásico es la implementación automática de operadores a partir de otros. Es muy utilizado en aritmética, pero también utilizable en otros tipos, como el siguiente ejemplo de Eli Bendersky[8]:

Listado 21.42: Ejemplo de CRTP como *mixin*.

```
1  template <typename Derived>
2  struct Comparisons { };
3
4  template <typename Derived>
5  bool operator==(const Comparisons<Derived>& o1, const Comparisons<
       Derived>& o2)
6  {
7      const Derived& d1 = static_cast<const Derived&>(o1);
8      const Derived& d2 = static_cast<const Derived&>(o2);
9
10     return !(d1 < d2) && !(d2 < d1);
11 }
12
13 template <typename Derived>
14 bool operator!=(const Comparisons<Derived>& o1, const Comparisons<
       Derived>& o2)
15 {
16     return !(o1 == o2);
17 }
```

Y con ello podemos definir todos los operadores de golpe sin más que definir *operator* <.

Listado 21.43: Ejemplo de *mixin* con CRTP para implementación automática de operadores.

```
1  class Person : public Comparisons<Person> {
2  public:
3      Person(string name_, unsigned age_)
4          : name(name_), age(age_) {}
5
6      friend bool operator<(const Person& p1, const Person& p2);
7  private:
8      string name;
```

[8] http://eli.thegreenplace.net/2011/05/17/the-curiously-recurring-template-pattern-in-c/

```
 9     unsigned age;
10 };
11
12 bool operator<(const Person& p1, const Person& p2) {
13     return p1.age < p2.age;
14 }
```

Consideraciones

La técnica que explota el patrón CRTP es denominada a veces como *polimorfismo estático*, por contraposición al dinámico de las funciones virtuales. La clase base utiliza la implementación correcta de los métodos redefinidos en las clases derivadas porque se le pasa como parámetro de plantilla. Esto es una ventaja y un inconveniente a la vez.

Por un lado la utilización de funciones no virtuales elimina las indirecciones y permite que sea lo más eficiente posible. Pero por otro lado no puede inferir el tipo de un objeto a través de un puntero a la clase base. Por ejemplo, si en el caso del visitante hubiera varios tipos derivados de `ObjectScene` y la clase `Scene` almacenara punteros a `ObjectScene`, el método `traverseObject()` no podría determinar qué función de visita debe invocar. La solución estándar en este caso sería emplear RTTI (*run-time type information*) para determinar el tipo de objeto en tiempo de ejecución, pero eso es mucho menos eficiente que las funciones virtuales.

Listado 21.44: Uso de RTTI para especializar la visita de objetos.

```
 1 void traverseObject(ObjectScene* o) {
 2     Character* c = dynamic_cast<Character*>(o);
 3     if (c) {
 4         getDerived().visitCharacter(c);
 5         return;
 6     }
 7     Weapon* w = dynamic_cast<Weapon*>(o);
 8     if (w) {
 9         getDerived().visitCharacter(w);
10         return;
11     }
12 }
```

21.2.4. Reactor

El patrón *Reactor* es un patrón arquitectural para resolver el problema de cómo atender peticiones concurrentes a través de señales y manejadores de señales.

Problema

Existen aplicaciones, como los servidores web, cuyo comportamiento es *reactivo*, es decir, a partir de la ocurrencia de un evento externo se realizan todas las operaciones necesarias para atender a ese evento externo. En el caso del servidor web, una conexión entrante (evento) dispararía la ejecución del código pertinente que crearía un hilo de ejecución para atender a dicha conexión. Pero también pueden tener comportamiento *proactivo*. Por ejemplo, una señal interna puede indicar cuándo destruir una conexión con un cliente que lleva demasiado tiempo sin estar accesible.

En los videojuegos ocurre algo muy similar: diferentes entidades pueden lanzar eventos que deben ser tratados en el momento en el que se producen. Por ejemplo, la pulsación de un botón en el joystick de un jugador es un evento que debe ejecutar el código pertinente para que la acción tenga efecto en el juego.

Solución

En el patrón Reactor se definen una serie de actores con las siguientes responsabilidades (véase figura 21.16):

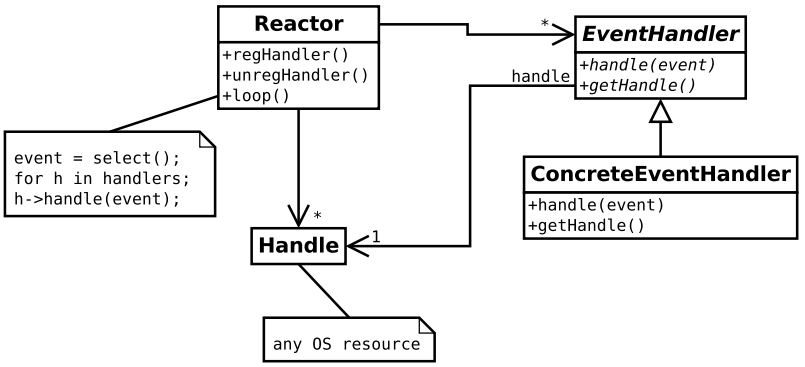


Figura 21.16: Diagrama de clases del patrón Reactor

- **Eventos**: los eventos externos que puedan ocurrir sobre los recursos (`Handles`). Normalmente su ocurrencia es asíncrona y siempre está relaciona a un recurso determinado.
- **Recursos** (`Handles`): se refiere a los objetos sobre los que ocurren los eventos.

  La pulsación de una tecla, la expiración de un temporizador o una conexión entrante en un socket son ejemplos de eventos que ocurren sobre ciertos recursos. La representación de los recursos en sistemas tipo GNU/Linux es el descriptor de fichero.
- **Manejadores de Eventos**: Asociados a los recursos y a los eventos que se producen en ellos, se encuentran los manejadores de eventos (`EventHandler`) que reciben una invocación a través del método `handle()` con la información del evento que se ha producido.
- **Reactor**: se trata de la clase que encapsula todo el comportamiento relativo a la desmultiplexación de los eventos en manejadores de eventos (*dispatching*). Cuando ocurre un cierto evento, se busca los manejadores asociados y se les invoca el método `handle()`.

En general, el comportamiento sería el siguiente:

1. Los manejadores se registran utilizando el método `regHandler()` del Reactor. De esta forma, el Reactor puede configurarse para esperar los eventos del recurso que el manejador espera. El manejador puede dejar de recibir notificaciones con `unregHandler()`.
2. A continuación, el Reactor entra en el bucle infinito (`loop()`), en el que se espera la ocurrencia de eventos.

3. Utilizando alguna llamada al sistema, como puede ser `select()`, el Reactor espera a que se produzca algún evento sobre los recursos monitorizados.
4. Cuando ocurre, busca los manejadores asociados a ese recurso y les invoca el método `handle()` con el evento que ha ocurrido como parámetro.
5. El manejador recibe la invocación y ejecuta todo el código asociado al evento.

Nótese que aunque los eventos ocurran concurrentemente el Reactor *serializa* las llamadas a los manejadores. Por lo tanto, la ejecución de los manejadores de eventos ocurre de forma secuencial.

Implementación

Desde el punto de vista de implementación un Reactor se comporta como un envoltorio orientado a objetos de los servicios de demultiplexión de eventos del sistema operativo. Ofrece una interfaz homogénea para llamadas al sistema tales como `select()`, `poll()`, o los *IO Completion Ports* de Windows.

Una implementación flexible de este patrón es la incluida en ACE (Adaptive Communications Environment) (*Adaptive Communications Environment*[9]). Éste sería un ejemplo mínimo con un manejador de eventos de teclado:

Listado 21.45: Ejemplo de uso de patrón *reactor*.

```
#include <ace/Reactor.h>

class MyEvHandler : public ACE_Event_Handler {
    virtual int handle_input (ACE_HANDLE h) {
        char buf[256];
        int n = ::read(h, buf, sizeof buf);
        if (n <= 0) return -1;
        // procesar buf ...
        return 0;
    }
};

int main (int argc, const char *argv[]) {
    ACE_Reactor reactor;
    MyEvHandler h;

    reactor.register_handler(ACE_STDIN,
                 &h,
                 ACE_Event_Handler::READ_MASK);
    for(;;)
        reactor.handle_events();

    return 0;
}
```

Para mezclar el bucle de eventos de Ogre con cualquier implementación del patrón reactor basta eliminar la llamada a `Root::startRendering()` e incluir dentro del bucle del Reactor una llamada a `Root::renderOneFrame()`.

[9] `http://www.cs.wustl.edu/~schmidt/ACE.html`

Consideraciones

Al utilizar un Reactor, se deben tener las siguientes consideraciones:

1. Los manejadores de eventos no pueden consumir mucho tiempo. Si lo hacen, pueden provocar un efecto *convoy* y, dependiendo de la frecuencia de los eventos, pueden hacer que el sistema sea inoperable. En general, cuanto mayor sea la frecuencia en que ocurren los eventos, menos tiempo deben consumir los manejadores.
2. Existen implementaciones de Reactors que permiten una desmultiplexación concurrente.
3. Desde un punto de vista general, el patrón Observer tiene un comportamiento muy parecido. Sin embargo, el Reactor está pensado para las relaciones 1 a 1 y no 1 a n como en el caso del Observer visto en el módulo 1.

21.2.5. Acceptor/Connector

Acceptor-Connector es un patrón de diseño propuesto por Douglas C. Schmidt [82] y utilizado extensivamente en ACE, su biblioteca de comunicaciones.

La mayoría de los videojuegos actuales necesitan comunicar datos entre jugadores de distintos lugares físicos. En toda comunicación en red intervienen dos ordenadores con roles bien diferenciados. Uno de los ordenadores toma el rol activo en la comunicación y solicita una conexión con el otro. El otro asume un rol pasivo esperando solicitudes de conexión. Una vez establecida la comunicación cualquiera de los ordenadores puede a su vez tomar el rol activo enviando datos o el pasivo, esperando la llegada de datos. Es decir, en toda comunicación aparece una fase de conexión e inicialización del servicio y un intercambio de datos según un patrón de intercambio de mensajes pre-establecido.

El patrón *acceptor-connector* se ocupa de la primera parte de la comunicación. Desacopla el establecimiento de conexión y la inicialización del servicio del procesamiento que se realiza una vez que el servicio está inicializado. Para ello intervienen tres componentes: *acceptors*, *connectors* y manejadores de servicio (*service handlers*. Un *connector* representa el rol activo, y solicita una conexión a un *acceptor*, que representa el rol pasivo. Cuando la conexión se establece ambos crean un manejador de servicio que procesa los datos intercambiados en la conexión.

Problema

El procesamiento de los datos que viajan por la red es en la mayoría de los casos independiente de qué protocolos, interfaces de programación de comunicaciones, o tecnologías específicas se utilicen para transportarlos. El establecimiento de la comunicación es un proceso inherentemente asimétrico (uno inicia la conexión mientras otro espera conexiones) pero una vez establecida la comunicación el transporte de datos es completamente ortogonal.

Desde el punto de vista práctico resuelve los siguientes problemas:

- Facilita el cambio de los roles de conexión sin afectar a los roles en el intercambio de datos.
- Facilita la adición de nuevos servicios y protocolos sin afectar al resto de la arquitectura de comunicación.

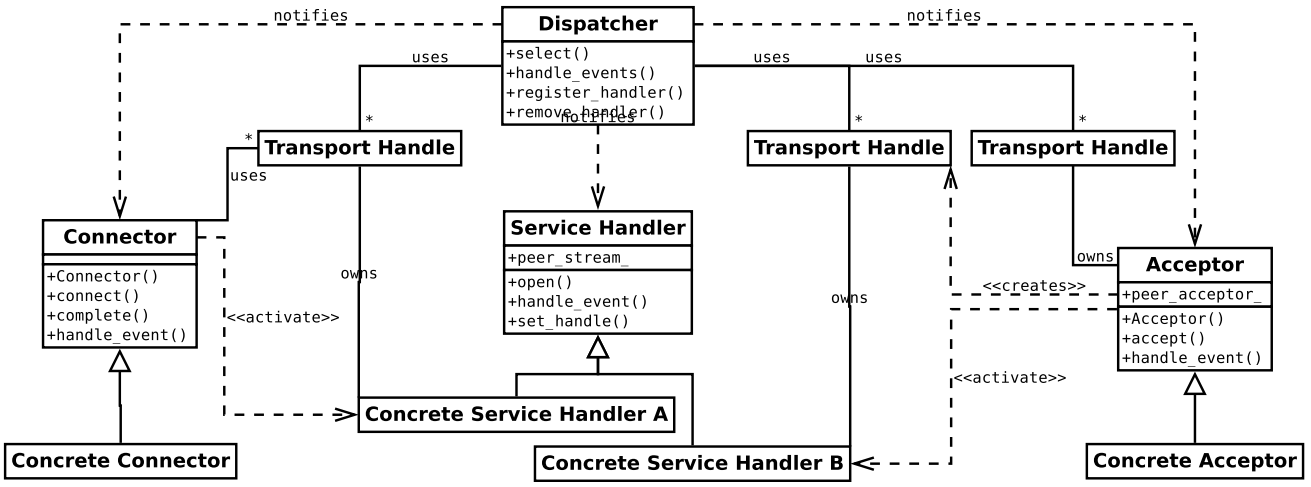


Figura 21.17: Estructura del patrón *acceptor-connector*.

- En los juegos de red a gran escala (MMORPG (Massively Multiplayer Online Role-Playing Game)) facilita la reducción de la latencia en el establecimiento de conexión usando mecanismos avanzados del sistema operativo, como conexiones asíncronas.

Solución

El funcionamiento es como sigue:

- Un `Acceptor` es una factoría que implementa el rol pasivo para establecer conexiones. Ante una conexión crea e inicializa un `Transport Handle` y un `Service Handler` asociados. En su inicialización, un `Acceptor` se asocia a una dirección de transporte (e.g. dirección IP y puerto TCP), y se configura para aceptar conexiones en modo pasivo. Cuando llega una solicitud de conexión realiza tres pasos:
  1. Acepta la conexión creando un `Transport Handle` que encapsula un extremo conectado.
  2. Crea un `Service Handler` que se comunicará directamente con el del otro extremo a través del `Transport Handle` asociado.
  3. Activa el `Service Handler` para terminar la inicialización.
- Un `Connector` es una factoría que implementa el rol activo de la conexión. En la inicialización de la conexión `connect()()` crea un `Transport Handle` que encapsula un extremo conectado con un `Acceptor` remoto, y lo asocia a un `Service Handler` preexistente.

World of Warcraft

WoW es el mayor MMORPG de la actualidad con más de 11.5 millones de suscriptores mensuales.

Tanto `Acceptor` como `Connector` pueden tener separadas las funciones de inicialización de la conexión de la función de completado de la conexión (cuando ya se tiene garantías de que el otro extremo ha establecido la conexión). De esta forma es fácil soportar conexiones asíncronas y síncronas de forma completamente transparente.

El `Dispatcher` es responsable de demultiplexar eventos del canal, tales como peticiones de conexión o peticiones de datos. Para el `Acceptor` demultiplexa indicaciones de conexión a través de los `Transport Handle`s que encapsulan direcciones de nivel de transporte. Para el `Connector` demultiplexa eventos de establecimiento de conexión que llegan cuando la solicitud de conexión es asíncrona.

El patrón *acceptor-connector* coopera perfectamente con el patrón *reactor*. Tanto el `Transport Handle` asociado al `Acceptor`, como el asociado al `Connector`, e incluso los asociados a los manejadores de servicio pueden ser un manejadores de eventos registrados en el *reactor* del sistema. De esta forma el `Dispatcher` pasa a ser un *reactor* que demultiplexa no solo eventos de red, sino de interfaz de usuario, o eventos del propio juego.

Implementación

Desde el punto de vista de la implementación, si nos restringimos a TCP y la API sockets el `Acceptor` no es más que una envoltura de la llamada al sistema `accept()`, el `Connector` una envoltura de la llamada al sistema `connect()`, y el `Dispatcher` o `Reactor` una envoltura de la llamada al sistema `select()` o `poll()`.

Una de las más flexibles implementaciones que existen de este patrón es la que ofrece ACE (*Adaptive Communications Environment*), biblioteca creada por el inventor del patrón y utilizada en multitud de sistemas de comunicaciones a escala global.

Otra implementación muy escalable y extremadamente elegante del patrón *acceptor-connector* es la incluida en la biblioteca ZeroC Ice, que ya conocemos. Sin embargo, el objeto de Ice es implementar un *middleware* de comunicaciones basado en el modelo de objetos distribuidos. Por tanto la implementación del patrón es privada, y no se expone a los usuarios. Ya examinaremos este modelo más adelante.

En ACE un servidor TCP mínimo atendiendo conexiones en el puerto 9999 tendría el siguiente aspecto:

Listado 21.46: Ejemplo de uso de patrón *acceptor-connector* (servidor).

```
1  #include <ace/SOCK_Acceptor.h>
2  #include <ace/Acceptor.h>
3  #include <ace/Svc_Handler.h>
4
5  class MySvcHandler : public ACE_Svc_Handler<ACE_SOCK_STREAM,
       ACE_MT_SYNCH> {
6      virtual int handle_input (ACE_HANDLE) {
7          char buf[256];
8          int n = peer().recv(buf, sizeof buf);
9          if (n <= 0) return -1;
10         // procesar buf ...
11         return 0;
12     }
13 };
14
15 typedef ACE_Acceptor <MySvcHandler, ACE_SOCK_ACCEPTOR> MyAcceptor;
16
17 int main (int argc, const char *argv[]) {
18     ACE_Reactor reactor;
19     MyAcceptor acceptor;
20
21     acceptor.open(ACE_INET_Addr(9999), &reactor);
22     for(;;) reactor.handle_events();
23     return 0;
24 }
```

Especializamos la plantilla del `Acceptor` con un `Svc_Handler` que tiene la lógica de intercambio de mensajes. Al instanciar el `Acceptor` le pasamos un `Reactor` para que automáticamente registre los nuevos `Svc_Handler` que crea en las nuevas conexiones.

El lado del cliente es muy similar, salvo que en este caso utilizamos un `Connector`.

Listado 21.47: Ejemplo de uso de patrón *acceptor-connector* (cliente).

```
#include <ace/SOCK_Connector.h>
#include <ace/Connector.h>
#include <ace/Svc_Handler.h>

class MySvcHandler : public ACE_Svc_Handler<ACE_SOCK_STREAM,
    ACE_MT_SYNCH> {
    virtual int handle_output (ACE_HANDLE) {
        char buf[]="Hello, World!\n";
        int n = peer().send(buf, sizeof buf);
        if (n <= 0) return -1;
        return 0;
    }
};

typedef ACE_Connector <MySvcHandler, ACE_SOCK_CONNECTOR>
    MyConnector;

int main (int argc, const char *argv[]) {
    ACE_Reactor reactor;
    MyConnector connector;
    MySvcHandler* psvc = 0;

    int n = connector.connect(psvc, ACE_INET_Addr(9999,"127.0.0.1")
        );
    if (n < 0) return 1;

    reactor.register_handler(psvc, ACE_Event_Handler::WRITE_MASK);
    for(;;)
        reactor.handle_events();

    return 0;
}
```

Como puede verse el `Connector` construye un `Svc_Handler` para procesar eventos. Nosotros registramos ese manejador en el reactor para generar mensajes hacia el servidor.

Téngase en cuenta que estos ejemplos son simples en exceso, con el propósito de ilustrar el uso del patrón. En un videojuego habría que tratar los errores adecuadamente y ACE permite también configurar el esquema de concurrencia deseado.

Consideraciones

Este patrón permite manejar de forma uniforme las comunicaciones multi-protocolo en juegos *online*. Además, coopera con el *reactor* de manera que podemos tener una única fuente de eventos en el sistema. Esto es muy interesante desde todos los puntos de vista, porque facilita enormemente la depuración, la síntesis de eventos en el sistema, la grabación de secuencias completas de eventos para su posterior reproducción, etc.

21.3. Programación genérica

La programación genérica es un paradigma de programación que trata de conseguir un mayor grado de reutilización tanto de las estructuras de datos como de los algoritmos, evitando así la duplicidad de código. Para lograrlo, los algoritmos deben escribirse evitando asociar los detalles a tipos de datos concretos. Por ejemplo, en un algoritmo de ordenación, la operación que compara dos elementos cualesquiera se delega a una entidad ajena al algoritmo: un operador de comparación.

Hoy en día, prácticamente todos los lenguajes de programación importantes disponen o han adoptado características de programación genérica (tal como los llamados «genéricos» en Java o C#).

El diseño de la librería STL pretende proporcionar herramientas básicas de programación genérica. No es casualidad que la creación de STL y las ideas tras el paradigma de la programación genérica fueran desarrolladas por los mismos autores, especialmente Alexander Stepanov y David Musser [65]. Y de ahí el interés por separar las estructuras de datos (los contenedores) de los algoritmos. Como veremos, los otros dos componentes de la STL (iteradores y functors) sirven también al mismo propósito: posibilitan la interacción entre contenedores y algoritmos, a la vez que permiten un acoplamiento mínimo.

Figura 21.18: Alexander Stepanov, padre de la programación genérica y la librería STL

Es interesante indicar que la disociación entre los datos y los algoritmos que los manejan contradice en cierta medida los principios de la programación orientada a objetos. En la POO las operaciones relativas a un tipo de dato concreto se ofrecen como métodos de dicha clase. El polimorfismo por herencia[10] permite en la práctica utilizar un algoritmo definido como un método de la superclase con instancias de sus subclases. Sin embargo, esto no se considera programación genérica pues la implementación del algoritmo normalmente depende al menos de la superclase de la jerarquía.

En STL los algoritmos están implementados normalmente como funciones (no métodos) y por supuesto no tienen estado, algo que por definición es ajeno a la POO. A pesar de ello, en el diseño de la librería están muy presentes los principios de orientación a objetos.

21.3.1. Algoritmos

Para conseguir estructuras de datos genéricas, los contenedores se implementan como plantillas —como ya se discutió en capítulos anteriores— de modo que el tipo de dato concreto que han de almacenar se especifica en el momento de la creación de la instancia.

Aunque es posible implementar algoritmos sencillos del mismo modo —parametrizando el tipo de dato— STL utiliza un mecanismo mucho más potente: los iteradores. Los iteradores permiten desacoplar tanto el tipo de dato como el modo en que se organizan y almacenan los datos en el contenedor.

Lógicamente, para que un algoritmo pueda hacer su trabajo tiene que asumir que tanto los elementos del contenedor como los iteradores tienen ciertas propiedades, o siendo más precisos, un conjunto de métodos con un comportamiento predecible. Por ejemplo, para poder comparar dos colecciones de elementos, deben ser comparables dos a dos para determinar su *igualdad* —sin entrar en qué significa eso realmente. Así

[10] también llamado polimorfismo «de subclase» o «de inclusión», en contraposición con el «polimorfismo paramétrico»

pues, el algoritmo `equal()` espera que los elementos soporten en «modelo» *EqualityComparable*, que implica que tienen sobrecargado el método `operator==()`, además de cumplir éste ciertas condiciones como reflexibidad, simetría, transitividad, etc.

algoritmos «escalares»

Aunque la mayoría de los algoritmos de la STL manejan secuencias delimitadas por dos iteradores, también hay algunos que utilizan datos escalares, tales como `min()`, `max()`, `power()` o `swap()` que pueden resultar útiles para componer algoritmos más complejos.

Escribiendo un algoritmo genérico

El mejor modo de comprender en qué consiste la «genericidad» de un algoritmo es crear uno desde cero. Escribamos nuestra propia versión del algoritmo genérico `count()` (uno de los más sencillos). Este algoritmo sirve para contar el número de ocurrencias de un elemento en una secuencia. Como una primera aproximación veamos cómo hacerlo para un array de enteros. Podría ser algo como:

Listado 21.48: Escribiendo un algoritmo genérico: `my_count()` (1/4)

```
1  int my_count1(const int* sequence, int size, int value) {
2    int retval = 0;
3    for (int i=0; i < size; ++i)
4      if (sequence[i] == value)
5        retval++;
6
7    return retval;
8  }
9
10 void test_my_count1() {
11   const int size = 5;
12   const int value = 1;
13   int numbers[] = {1, 2, 3, 1, 2};
14
15   assert(my_count1(numbers, size, value) == 2);
16 }
```

Destacar el especificador `const` en el parámetro `sequence` (línea 1). Le indica al compilador que esta función no modificará el contenido del array. De ese modo es más general; se podrá aplicar a cualquier array (sea constante o no).

Recuerda, en las funciones, aquellos parámetros que no impliquen copia (puntero o referencia) deberían ser constantes si la función efectivamente no va a modificarlos.

En la siguiente versión vamos a cambiar la forma de iterar sobre el array. En lugar de emplear un índice vamos a utilizar un puntero que se desplaza a través del array. Esta versión mantiene el prototipo, es decir, se invoca de la misma forma.

Listado 21.49: Escribiendo un algoritmo genérico: `my_count()` (2/4)

```
1  int my_count2(const int* first, int size, int value) {
2    int retval = 0;
3    for (const int* it=first; it < first + size; ++it)
4      if (*it == value)
5        retval++;
6
7    return retval;
8  }
```

Dos cuestiones a destacar:

- Utiliza aritmética de punteros. Es decir, la dirección del puntero `it` (linea 3) no se incrementa de uno en uno, sino que depende del tamaño del tipo `int`.
- El valor consultado en el array se obtiene de-referenciando el puntero (`*it` en la línea 4).

A continuación la función cambia para imitar la signatura habitual de STL. En lugar de pasar un puntero al comienzo y un tamaño, se le pasan punteros al comienzo y al final-más-uno.

Listado 21.50: Escribiendo un algoritmo genérico: `my_count()` (3/4)

```
1  int my_count3(const int* first, const int* last, int value) {
2    int retval = 0;
3    for (const int* it=first; it < last; ++it)
4      if (*it == value)
5        retval++;
6
7    return retval;
8  }
9
10 void test_my_count3() {
11   const int size = 5;
12   const int value = 1;
13   int numbers[] = {1, 2, 3, 1, 2};
14
15   assert(my_count3(numbers, numbers+size, value) == 2);
16 }
```

Se puede apreciar como el criterio del final-mas-uno simplifica la invocación, puesto que el valor correcto se consigue con `numbers+size` (línea 15) y la condición de parada es también más simple (`it<last`) en la línea 3.

Por último, veamos como queda la función cambiando los punteros por iteradores. Es fácil comprobar como resultan funcionalmente equivalentes, hasta el punto de que la función se puede utilizar también con un contenedor `vector`. También se ha convertido la función en una plantilla, de modo que se podrá utilizar con cualquier tipo de dato, a condición de que sus elementos soporten la operación de comprobación de igualdad:

Listado 21.51: Escribiendo un algoritmo genérico: `my_count()` (4/4)

```
1  template <typename Iter, typename T>
2  int my_count4(Iter first, Iter last, T value) {
3    int retval = 0;
4    for (Iter it=first; it < last; ++it)
5      if (*it == value)
6        retval++;
7
8    return retval;
9  }
10
11 void test_my_count4_numbers() {
12   const int size = 5;
13   const int value = 1;
14   int numbers[] = {1, 2, 3, 1, 2};
15   vector<int> numbers_vector(numbers, numbers + size);
16
17   assert(my_count4(numbers, numbers+size, value) == 2);
18   assert(my_count4(numbers_vector.begin(), numbers_vector.end(),
19                    value) == 2);
```

```
20 }
21
22 void test_my_count4_letters() {
23   const int size = 5;
24   const int value = 'a';
25   char letters[] = {'a', 'b', 'c', 'a', 'b'};
26   vector<char> letters_vector(letters, letters + size);
27
28   assert(my_count4(letters, letters+size, value) == 2);
29   assert(my_count4(letters_vector.begin(), letters_vector.end(),
30                    value) == 2);
31 }
```

Esta última versión es bastante similar a la implementación habitual del algoritmo `count()` estándar con la salvedad de que éste último realiza algunas comprobaciones para asegurar que los iteradores son válidos.

Comprobamos que nuestras funciones de prueba funcionan exactamente igual utilizando el algoritmo `count()` estándar[11]:

Listado 21.52: El algoritmo `count()` estándar se comporta igual

```
1  void test_count_numbers() {
2    const int size = 5;
3    const int value = 1;
4    int numbers[] = {1, 2, 3, 1, 2};
5    vector<int> numbers_vector(numbers, numbers + size);
6
7    assert(count(numbers, numbers+size, value) == 2);
8    assert(count(numbers_vector.begin(), numbers_vector.end(),
9             value) == 2);
10 }
11
12 void test_count_letters() {
13   const int size = 5;
14   const int value = 'a';
15   char letters[] = {'a', 'b', 'c', 'a', 'b'};
16   vector<char> letters_vector(letters, letters + size);
17
18   assert(count(letters, letters+size, value) == 2);
19   assert(count(letters_vector.begin(), letters_vector.end(),
20            value) == 2);
21 }
```

21.3.2. Predicados

En el algoritmo `count()`, el criterio para contar es la igualdad con el elemento proporcionado. Eso limita mucho sus posibilidades porque puede haber muchos otros motivos por los que sea necesario contar elementos de una secuencia: esferas de color rojo, enemigos con nivel mayor al del jugador, armas sin munición, etc.

Lógica de predicados

En lógica, un **predicado** es una expresión que se puede evaluar como cierta o falsa en función del valor de sus variables de entrada. En programación, y en particular en la STL, un predicado es una función (en el sentido amplio) que acepta un valor del mismo tipo que los elementos de la secuencia sobre la que se usa y devuelve un valor booleano.

Por ese motivo, muchos algoritmos de la STL tienen una versión alternativa que permite especificar un parámetro adicional llamado *predicado*. El algoritmo invocará el predicado para averiguar si se cumple la condición indicada por el programador y así determinar cómo debe proceder con cada elemento de la secuencia.

En C/C++, para que una función pueda invocar a otra (en este caso, el algoritmo al predicado) se le debe pasar como parámetro un puntero a función.

[11] Para utilizar los algoritmos estándar se debe incluir el fichero `<algorithm>`.

Veamos la definición de un predicado (`not_equal_2`) que, como habrá imaginado, será cierto para valores distintos a 2:

Listado 21.53: Predicado `not_equal_2()`

```
1  bool not_equal_2(int n) {
2    return n != 2;
3  }
4
5  void test_not_equal_2() {
6    const int size = 5;
7    int numbers[] = {1, 2, 3, 1, 2};
8
9    assert(count_if(numbers, numbers+size, not_equal_2) == 3);
10 }
```

Igual que con cualquier otro tipo de dato, cuando se pasa un *puntero a función* como argumento, el parámetro de la función que lo acepta debe estar declarado con ese mismo tipo. Concretamente el tipo del predicado `not_equal_2` sería algo como:

Listado 21.54: Tipo para un predicado que acepta un argumento entero

```
1 bool (*)(int);
```

El algoritmo `count_if()` lo acepta sin problema. Eso se debe a que, como ya hemos dicho, los algoritmos son funciones-plantilla y dado que la secuencia es un array de enteros, asume que el valor que acepta el predicado debe ser también un entero, es decir, el algoritmo determina automáticamente la signatura del predicado.

Aunque funciona, resulta bastante limitado. No hay forma de modificar el comportamiento del predicado en tiempo de ejecución. Es decir, si queremos contar los elementos distintos de 3 en lugar de 2 habría que escribir otro predicado diferente. Eso es porque el único argumento que puede tener el predicado es el elemento de la secuencia que el algoritmo le pasará cuando lo invoque[12], y no hay modo de darle información adicional de forma limpia.

21.3.3. Functors

Existe sin embargo una solución elegante para conseguir «predicados configurables». Consiste es declarar una clase que sobrecargue el operador de invocación — método `operator()()` — que permite utilizar las instancias de esa clase como si fueran funciones. Las clases que permiten este comportamiento se denominan «functors»[13]. Veamos como implementar un predicado `not_equal()` como un functor:

Listado 21.55: Predicado `not_equal()` para enteros (como functor)

```
1  class not_equal {
2    const int _ref;
3
4  public:
5    not_equal(int ref) : _ref(ref) {}
6
7    bool operator()(int value) {
8      return value != _ref;
9    }
10 };
```

[12] En la terminología de STL se denomina «predicado unario»

[13] «functor» se traduce a veces como «objeto-función».

Y dos pruebas que demuestran su uso:

Listado 21.56: Ejemplos de uso de `not_equal()`

```
1  void test_not_equal_functor() {
2    not_equal not_equal_2(2);
3
4    assert(not_equal_2(0));
5    assert(not not_equal_2(2));
6  }
7
8  void test_not_equal_count_if() {
9    const int size = 5;
10   int numbers[] = {1, 2, 3, 1, 2};
11
12   assert(count_if(numbers, numbers+size, not_equal(2)) == 3);
13 }
```

Para disponer de un predicado lo más flexible posible deberíamos implementarlo como una clase plantilla de modo que sirva no solo para enteros:

Listado 21.57: Predicado `not_equal()` genérico como functor

```
1  template <typename _Arg>
2  class not_equal {
3    const _Arg _ref;
4
5  public:
6    not_equal(_Arg ref) : _ref(ref) {}
7
8    bool operator()(_Arg value) const {
9      return value != _ref;
10   }
11 };
```

Pero los predicados no son la única utilidad interesante de los functors. Los predicados son una particularización de las «funciones» (u operadores). Los operadores pueden devolver cualquier tipo. no sólo booleanos.

La STL clasifica los operadores en 3 categorías básicas:

Generador Una función que no acepta argumentos.

Unario Una función que acepta un argumento.

Binario Una función que acepta dos argumentos.

Aunque obviamente puede definirse un operador con 3 o más argumentos, no hay ningún algoritmo estándar que los utilice. Si el functor devuelve un booleano es cuando se denomina «predicado unario» o «binario» respectivamente. Para ser un predicado debe tener al menos un argumento como hemos visto. Además se habla también de modalidades «adaptables» para las tres categorías, que se distinguen porque exponen los tipos de sus argumentos y valor de retorno como atributos de la clase. Los veremos más adelante.

Los operadores (los functors que no son predicados) se utilizan normalmente en algoritmos que realizan algún cálculo con los elementos de una secuencia.

Como ejemplo, el siguiente listado multiplica los elementos del array `numbers`:

Listado 21.58: `accumulate()` multiplicando los elementos de una secuencia de enteros

```
1 void test_accumulate_multiplies() {
2   int numbers[] = {1, 2, 3, 4};
3   const int size = sizeof(numbers) / sizeof(int);
4
5   int result = accumulate(numbers, numbers+size,
6                           1, multiplies<int>());
7   assert(result == 24);
8 }
```

El algoritmo `accumulate()` aplica el functor binario especificado como último parámetro (`multiplies()` en el ejemplo) empezando por el valor inicial indicado como tercer parámetro (1) y siguiendo con los elementos del rango especificado. Corresponde con la operación $\prod_{i=1}^{n} i$.

Además de `multiplies()`, la librería estándar incluye muchos otros functors que se clasifican en operaciones aritméticas (grupo al que corresponde `multiplies()`), lógicas, de identidad y comparaciones. Los iremos viendo y utilizando a lo largo de esta sección.

21.3.4. Adaptadores

Es habitual que la operación que nos gustaría que ejecute el algoritmo sea un método (una función miembro) en lugar de una función estándar. Si tratamos de pasar al algoritmo un puntero a método no funcionará, porque el algoritmo no le pasará el parámetro implícito `this` que todo método necesita.

Una posible solución sería escribir un functor que invoque el método deseado, como se muestra en el siguiente listado.

Listado 21.59: Adaptador «manual» para un método

```
1 class Enemy {
2 public:
3   bool is_alive(void) const {
4     return true;
5   }
6 };
7
8 class enemy_alive {
9 public:
10  bool operator()(Enemy enemy) {
11    return enemy.is_alive();
12  }
13 };
14
15 void test_my_adapter() {
16   vector<Enemy> enemies(2);
17
18   assert(count_if(enemies.begin(), enemies.end(),
19                   enemy_alive()) == 2);
20 }
```

Listado 21.60: Uso del adaptador `mem_fun_ref()`

```
1 void test_mem_fun_ref() {
2   vector<Enemy> enemies(2);
3
```

`mem_fun()`

Existe un adaptador alternativo llamado `mem_fun()` que debe utilizarse si los elementos del contenedor son punteros. Si son objetos o referencias se utiliza `mem_fun_ref()`.

```
4   assert(count_if(enemies.begin(), enemies.end(),
5                   mem_fun_ref(&Enemy::is_alive)) == 2);
6 }
```

Veamos de nuevo el problema de tener una operación o predicado que requiere un argumento adicional aparte del elemento de la secuencia. En la sección 21.3.3 resolvimos el problema creando un functor (`not_equal`) que sirviera como adaptador. Bien, pues eso también lo prevé la librería y nos proporciona dos adaptadores llamados `bind1st()` y `bind2st()` para realizar justo esa tarea, y de manera genérica. Veamos cómo —gracias a `bind2nd()`— es posible reescribir el listado 21.53 de modo que se puede especificar el valor con el que comparar (parámetro `ref`) sin tener que escribir un functor ad-hoc:

Listado 21.61: Uso del adaptador `bind2nd()`

```
1 bool not_equal(int n, int ref) {
2   return n != ref;
3 }
4
5 void test_not_equal_bind() {
6   const int size = 5;
7   int numbers[] = {1, 2, 3, 1, 2};
8
9   assert(count_if(numbers, numbers+size,
10                  bind2nd(ptr_fun(not_equal), 2)) == 3);
11 }
```

Nótese que `bind2nd()` espera un functor como primer parámetro. Como lo que tenemos es una función normal es necesario utilizar otro adaptador llamado `ptr_fun()`, que como su nombre indica adapta un puntero a función a functor.

`bind2nd()` pasa su parámetro adicional (el 2 en este caso) como segundo parámetro en la llamada a la función `not_equal()`, es decir, la primera llamada para la secuencia del ejemplo será `not_equal(1, 2)`. El primer argumento (el 1) es el primer elemento obtenido de la secuencia. El adaptador `bind1st()` los pasa en orden inverso, es decir, pasa el valor extra como primer parámetro y el elemento de la secuencia en segunda posición.

Hay otros adaptadores de menos uso:

`not1()` devuelve un predicado que es la negación lógica del predicado *unario* al que se aplique.

`not2()` devuelve un predicado que es la negación lógica del predicado *binario* al que se aplique.

`compose1()` devuelve un operador resultado de componer las dos funciones unarias que se le pasan como parámetros. Es decir, dadas las funciones $f(x)$ y $g(x)$ devuelve una función $f(g(x))$.

`compose2()` devuelve un operador resultado de componer una función binaria y dos funciones unarias que se le pasan como parámetro del siguiente modo. Dadas las funciones $f(x, y)$, $g_1(x)$ y $g_2(x)$ devuelve una función $h(x) = f(g_1(x), g_2(x))$.

Nomenclatura

Los nombres de los algoritmos siguen ciertos criterios. Como ya hemos visto, aquellos que tienen una versión acabada en el sufijo `_if` aceptan un predicado en lugar de utilizar un criterio implícito. Los que tienen el sufijo `_copy` generan su resultado en otra secuencia, en lugar de modificar la secuencia original. Y por último, los que acaban en `_n` aceptan un iterador y un entero en lugar de dos iteradores; de ese modo se pueden utilizar para insertar en «cosas» distintas de contenedores, por ejemplo flujos.

21.3.5. Algoritmos idempotentes

Los algoritmos idempotentes (*non-mutating*) son aquellos que no modifican el contenedor sobre el que se aplican. Podríamos decir que son algoritmos *funcionales* desde el punto de vista de ese paradigma de programación. Nótese que aunque el algoritmo en sí no afecte al contenedor, las operaciones que se realicen con él sí pueden modificar los objetos contenidos.

`for_each()`

El algoritmo `for_each()` es el más simple y general de la STL. Es equivalente a un bucle `for` convencional en el que se ejecutara un método concreto (o una función independiente) sobre cada elemento de un rango. Veamos un ejemplo sencillo en el que se recargan todas las armas de un jugador:

Listado 21.62: Ejemplo de uso del algoritmo `for_each()`

```
1  class Weapon {
2  public:
3    void reload() { /* some code */ }
4  };
5
6  void test_for_each() {
7    vector<Weapon> weapons(5);
8    for_each(weapons.begin(), weapons.end(),
9             mem_fun_ref(&Weapon::reload));
10 }
```

`find()` / `find_if()`

Devuelve un iterador al primer elemento del rango que coincide con el indicado (si se usa `find()`) o que cumple el predicado (si se usa `find_if()`). Devuelve el iterador `end` si no encuentra ninguna coincidencia. Un ejemplo en el que se busca el primer entero mayor que 6 que haya en el rango:

Listado 21.63: Ejemplo de `find_if()`

```
1  void test_find_if() {
2    const int size = 5;
3    const int value = 1;
4    int numbers[] = {2, 7, 12, 9, 4};
5
6    assert(find_if(numbers, numbers + size,
7                   bind2nd(greater<int>(), 6)) == numbers+1);
8  }
```

Se utiliza `greater`, un predicado binario que se cumple cuando su primer parámetro (el elemento del rango) es mayor que el segundo (el 6 que se pasa `bind2nd()`). El resultado del algoritmo es un iterador al segundo elemento (el 7) que corresponde con `numbers+1`. Hay algunos otros functors predefinidos para comparación: `equal_to()`, `not_equal_to()`[14], `less()`, `less_equal()` y `greater_equal()`.

[14] Equivalente al que implementamos en el listado 21.57.

count() / count_if()

Como ya hemos visto en ejemplos anteriores `count()` devuelve la cantidad de elementos del rango igual al dado, o que cumple el predicado, si se usa la modalidad `count_if()`.

mismatch()

Dados dos rangos, devuelve un par de iteradores a los elementos de cada rango en el que las secuencias difieren. Veamos el siguiente ejemplo —extraído de la documentación de SGI[15]:

Listado 21.64: Ejemplo de uso de `mismatch()`

```
1 void test_mismatch() {
2   int A1[] = { 3, 1, 4, 1, 5, 9, 3};
3   int A2[] = { 3, 1, 4, 2, 8, 5, 7};
4   const int size = sizeof(A1) / sizeof(int);
5
6   pair<int*, int*> result = mismatch(A1, A1 + size, A2);
7   assert(result.first == A1 + 3);
8   assert((*result.first) == 1 and (*result.second) == 2);
9 }
```

Muchos algoritmos de transformación que manejan dos secuencias requieren solo *tres* iteradores. El tercer iterador indica el comienzo de la secuencia de salida y se asume que ambas secuencias son del mismo tamaño.

equal()

Indica si los rangos indicados son iguales. Por defecto utiliza el `operator==()`, pero opcionalmente es posible indicar un predicado binario como cuarto parámetro para determinar en qué consiste la «igualdad». Veamos un ejemplo en el que se comparan dos listas de figuras que se considerarán iguales simplemente porque coincida su color:

Listado 21.65: Ejemplo de uso de `equal()`

```
1  enum Color{BLACK, WHITE, RED, GREEN, BLUE};
2
3  class Shape {
4  public:
5    Color color;
6
7    Shape(void) : color(BLACK) {}
8    bool cmp(Shape other) {
9      return color == other.color;
10   }
11 };
12
13 void test_equal() {
14   const int size = 5;
```

[15] http://www.sgi.com/tech/stl/mismatch.html

```
15   Shape shapes1[size], shapes2[size];
16   shapes2[3].color = RED;
17
18   assert(equal(shapes1, shapes1+size, shapes2,
19            mem_fun_ref(&Shape::cmp)) == false);
20 }
```

`search()`

Localiza la posición del segundo rango en el primero. Devuelve un iterador al primer elemento. Opcionalmente acepta un predicado binario para especificar la igualdad entre dos elementos. Veamos este ejemplo extraído de la documentación de SGI.

Listado 21.66: Ejemplo de uso de `search()`

```
1 void test_search() {
2   const char s1[] = "Hello, world!";
3   const char s2[] = "world";
4   const int n1 = strlen(s1);
5   const int n2 = strlen(s2);
6
7   const char* p = search(s1, s1 + n1, s2, s2 + n2);
8   assert(p == s1 + 7);
9 }
```

El algoritmo `find_end()` (a pesar de su nombre) es similar a `search()` solo que localiza la última aparición en lugar de la primera.

El algoritmo `search_n()` también es similar. Busca una secuencia de n elementos iguales (no otro rango) que debería estar contenida en el rango indicado.

21.3.6. Algoritmos de transformación

Normalmente, en los algoritmos de transformación (*mutating algorithms*) se distingue entre el rango o secuencia de entrada y la de salida, ya que su operación implica algún tipo de modificación (inserción, eliminación, cambio, etc.) sobre los elementos de la secuencia de salida.

Es importante recordar que las secuencias de salida que se utilizan en los algoritmos de transformación deben disponer de memoria suficiente para los datos que recibirán u obtendremos comportamientos erráticos aleatorios y errores de acceso a memoria en tiempo de ejecución (SEGFAULT).

`copy()`

Copia los elementos de un rango en otro. No debería utilizarse para copiar una secuencia completa en otra ya que el operador de asignación que tienen todos los contenedores resulta más eficiente. Sí resulta interesante para copiar fragmentos de secuencias. Veamos un uso interesante de `copy()` para enviar a un flujo (en este caso `cout`) el contenido de una secuencia.

Listado 21.67: Ejemplo de uso de `copy()`

```
int main() {
  int values[] = {1, 2, 3, 4, 5};
  const int size = sizeof(values) / sizeof(int);

  copy(values+2, values+size,
       ostream_iterator<int>(cout, ", "));
  cout << endl;
}
```

La plantilla `ostream_iterator` devuelve un iterador de inserción para un tipo concreto (`int` en el ejemplo) que escribirá en el flujo (`cout`) los elementos que se le asignen, escribiendo además una cadena opcional después de cada uno (una coma).

Existe una variante llamada `copy_backward()` que copia desde el final y en la que se debe pasar un iterador de la secuencia de salida al que copiar el último elemento.

`swap_ranges()`

Intercambia el contenido de dos secuencias. Como es habitual, se pasan los iteradores a principio y fin de la primera secuencia y al principio de la segunda, dado que asume que los rangos deben ser del mismo tamaño. Nótese que este algoritmo modifica ambos rangos.

`transform()`

El algoritmo `transform()` es uno de los más versátiles de la librería. La versión básica (que opera sobre un único rango de entrada) aplica un operador unario a cada elemento del rango y escribe el resultado a un iterador de salida.

Existe una versión alternativa (sobrecargada) que acepta dos secuencias de entrada. En este caso, el algoritmo utiliza un operador binario al que pasa un elemento que obtiene de cada una de las secuencias de entrada, el resultado se escribe sobre el iterador de salida.

Es interesante destacar que en ambos casos, el iterador de salida puede referirse a una de las secuencias de entrada.

Veamos un ejemplo en el que se concatenan las cadenas de dos vectores y se almacenan en un tercero haciendo uso del operador `plus()`:

Listado 21.68: Ejemplo de uso de `transform()`

```
void test_transform() {
  vector<string> v1, v2, result(2);
  v1.push_back("hello ");
  v1.push_back("bye ");
  v2.push_back("world");
  v2.push_back("hell");

  transform(v1.begin(), v1.end(), v2.begin(),
            result.begin(),
            plus<string>());

  assert(result[0] == "hello world");
  assert(result[1] == "bye hell");
}
```

`replace()` / `replace_if()`

Dado un rango, un valor antiguo y un valor nuevo, substituye todas las ocurrencias del valor antiguo por el nuevo en el rango. La versión `replace_if()` substituye los valores que cumplan el predicado unario especificado por el valor nuevo. Ambos utilizan un única secuencia, es decir, hacen la substitución in situ.

Existen variantes llamadas `replace_copy()` y `replace_copy_if()` respectivamente en la que se copian los elementos del rango de entrada al de salida a la vez que se hace la substitución. En este caso la secuencia original no cambia.

`fill()`

Dado un rango y un valor, copia dicho valor en todo el rango:

Listado 21.69: Ejemplo de uso de `fill()`

```
1 void test_fill() {
2   vector<float> v(10);
3   assert(count(v.begin(), v.end(), 0));
4
5   fill(v.begin(), v.end(), 2);
6   assert(count(v.begin(), v.end(), 2) == 10);
7 }
```

La variante `fill_n()` utiliza un único iterador de salida y copia sobre él n copias del valor especificado. Útil con iteradores de inserción.

`generate()`

En realidad es una variante de `fill()` salvo que los valores los obtiene de un operador que se le da como parámetro, en concreto un «generador», es decir, una función/functor sin parámetros que devuelve un valor:

Listado 21.70: Ejemplo de uso de `generate()`

```
1  class next {
2    int _last;
3  public:
4    next(int init) : _last(init) {}
5    int operator()() {
6      return _last++;
7    }
8  };
9
10 void test_generate() {
11   vector<int> v(10);
12   generate(v.begin(), v.end(), next(10));
13   assert(v[9] == 19);
14 }
```

Existe un algoritmo `generate_n()` al estilo de `copy_n()` o `fill_n()` que en lugar de dos iteradores, espera un iterador y una cantidad de elementos a generar.

remove()

Dado un rango y un valor, elimina todas las ocurrencias de dicho valor y retorna un iterador al nuevo último elemento. En realidad `remove()` no elimina nada, solo reordena la secuencia, colocando los elementos «eliminados» detrás del iterador que retorna.

Como en el caso de `replace()` existen alternativas análogas llamadas `replace_if()`, `replace_copy()` y `replace_copy_if()`.

El siguiente listado demuestra la diferencia que supone para el contenedor usar el algoritmo `remove()` respecto a usar además el método `erase()`. En la función `test_remove_and_erase()` se utiliza el iterador devuelto por `remove()` para eliminar los elementos del contenedor. Nótese que incluso en este caso lo que cambia es el tamaño del contenedor (`size`), pero no su capacidad (`capacity`).

Listado 21.71: Ejemplo de uso de `remove()`

```
1  const int size = 8;
2  int numbers[] = {1, 2, 3, 1, 4, 5, 1, 6};
3
4  void test_remove() {
5    vector<int> numbers_vector(numbers, numbers + size);
6
7    vector<int>::iterator it = \
8      remove(numbers_vector.begin(), numbers_vector.end(), 1);
9
10   assert(it == numbers_vector.begin() + 5);
11   assert(numbers_vector.size() == 8);
12 }
13
14 void test_remove_and_erase() {
15   vector<int> numbers_vector(numbers, numbers + size);
16
17   vector<int>::iterator it = \
18     remove(numbers_vector.begin(), numbers_vector.end(), 1);
19
20   numbers_vector.erase(it, numbers_vector.end());
21
22   assert(numbers_vector.size() == 5);
23 }
```

Esta combinación de `remove()` and `erase()` es una expresión idiomática (*idiom*) habitual entre los usuarios de STL.

unique()

Elimina elementos duplicados consecutivos. Dado que puede eliminar elementos in situ, retorna un iterador al nuevo último elemento de la secuencia. Existe una modalidad `unique_copy()` que copia el resultado sobre un iterador de salida dejando al secuencia original intacta. En ambos casos existen también modalidades que aceptan un predicado binario para definir la «igualdad» entre elementos.

reverse()

Invierte un rango in situ. También existe una modalidad que deja la secuencia original intacta llamada `reverse_copy()` . Se ilustra con un sencillo ejemplo que invierte **parte** de una cadena y no el contenedor completo:

Listado 21.72: Ejemplo de uso de `reverse()`

```
1 void test_reverse() {
2   char word[] = "reversion";
3   const int size = strlen(word);
4
5   reverse(word + 5, word + size);
6
7   assert(strcmp(word, "revernois") == 0);
8 }
```

`rotate()`

Rota los elementos del rango especificado por 3 iteradores, que indican el inicio, el punto medio y el final del rango. Existe una modalidad `rotate_copy()`, que como siempre aplica el resultado a un iterador en lugar de modificar el original.

Algoritmos aleatorios

Hay tres algoritmos que tienen que ver con operaciones aleatorias sobre una secuencia:

`random_shuffle()` reordena de forma aleatoria los elementos del rango.

`random_sample()` elige aleatoriamente elementos de la secuencia de entrada y los copia en la secuencia de salida. Es interesante destacar que este algoritmo requiere 4 iteradores ya que se puede crear una secuencia de salida de tamaño arbitrario, siempre que sea menor o igual que la secuencia de entrada.

`random_shuffle_n()` realiza la misma operación que `random_sample()` salvo que la cantidad de elementos a generar se especifica explícitamente en lugar de usar un cuarto iterador. Eso permite utilizarlo con un iterador de inserción.

Los tres aceptan opcionalmente una función que genere números aleatorios.

`partition()`

Dada una secuencia y un predicado, el algoritmo reordena los elementos de modo que los que satisfacen el predicado aparecen primero y los que lo incumplen después. Devuelve un iterador al primer elemento que incumple el predicado.

La modalidad `stable_partition()` preserva el orden de los elementos en cada parte respecto al orden que tenían en la secuencia original.

21.3.7. Algoritmos de ordenación

Los algoritmos de ordenación también son de transformación, pero se clasifican como un grupo distinto dado que todos tiene que ver con la ordenación de secuencias u operaciones con secuencias ordenadas.

`sort()`

Ordena in situ el rango especificado por dos iteradores. La modalidad `stable_sort()` preserva el orden relativo original a costa de algo menos de rendimiento. Veamos un ejemplo sencillo tomado del manual de SGI para ordenar un array de caracteres:

Listado 21.73: Ejemplo de uso de `sort()`

```
1  bool less_nocase(char c1, char c2) {
2    return tolower(c1) < tolower(c2);
3  }
4
5  void test_sort() {
6    char letters[] = "ZfdBeACFDbEacz";
7    const int size = strlen(letters);
8
9    sort(letters, letters+size, less_nocase);
10
11   char expected[] = "AaBbCcdDeEfFZz";
12   assert(equal(letters, letters+size, expected));
13 }
```

La mayoría de los algoritmos de ordenación aceptan un predicado especial para comparación de elementos dos a dos. Es muy habitual ordenar secuencias de elementos no numéricos o por características que tienen poco que ver con la relación mayor o menor en el sentido tradicional del término.

El algoritmo `partial_sort()` ordena parcialmente una secuencia especificada por tres iteradores de modo que solo el rango correspondiente a los dos primeros estará ordenado en la secuencia resultante. Tiene una modalidad `partial_sort_copy()` que no modifica la secuencia original.

`nth_element()`

Dada una secuencia y tres iteradores, ordena la secuencia de modo que todos los elementos en el subrango por debajo del segundo iterador (`nth`) son menores que los elementos que quedan por encima. Además, el elemento apuntado por el segundo iterador es el mismo que si se hubiera realizado una ordenación completa.

Operaciones de búsqueda

A continuación se incluye una pequeña descripción de los algoritmos relacionados con búsquedas binarias:

`binary_search()` determina si el valor indicado se encuentra en la secuencia.

`lower_bound()` devuelve un iterador a la primera posición en la que es posible insertar el elemento indicado manteniendo el orden en la secuencia.

`upper_bound()` devuelve un iterador a la última posición en la que es posible insertar el elemento indicado manteniendo el orden en la secuencia.

`equal_range()` combina los dos algoritmos anteriores. Devuelve un par con los iteradores a la primera y última posición en la que es posible insertar el elemento indicado manteniendo el orden de la secuencia.

Se muestra un ejemplo de los cuatro algoritmos:

Listado 21.74: Ejemplo de uso de los algoritmos de búsqueda

```
1  int numbers[] = {0, 3, 7, 7, 10, 11, 15};
2  int size = sizeof(numbers) / sizeof(int);
3
4  void test_binary_search() {
```

```
5   assert(binary_search(numbers, numbers+size, 6) == false);
6   assert(binary_search(numbers, numbers+size, 10));
7 }
8
9 void test_bounds() {
10  assert(lower_bound(numbers, numbers+size, 6) == numbers+2);
11  assert(upper_bound(numbers, numbers+size, 8) == numbers+4);
12 }
13
14 void test_equal_range() {
15  pair<int*, int*> bounds = equal_range(numbers, numbers+size, 7);
16  assert(bounds.first == numbers+2 and bounds.second == numbers+4);
17 }
```

merge() combina dos secuencias, dadas por dos pares de iteradores, y crea una tercera secuencia que incluye los elementos de ambas, manteniendo el orden.

Mínimo y máximo

Los algoritmos `min_element()` y `max_element()` permiten obtener respectivamente el elemento mínimo y máximo del rango especificado. Veamos un ejemplo:

Listado 21.75: Ejemplo de uso de `max_element()` y `min_element()`

```
1 char letters[] = "ZfdBeACFDbEacz";
2 const int size = strlen(letters);
3
4 void test_min() {
5   char* result = min_element(letters, letters+size);
6   assert(*result == 'A');
7 }
8
9 void test_max() {
10  char* result = max_element(letters, letters+size);
11  assert(*result == 'z');
12 }
```

21.3.8. Algoritmos numéricos

accumulate()

Aplica un operador (la suma si no se especifica otro) sobre el rango especificado por dos iteradores. Debe indicarse también un valor inicial ya que el algoritmo opera sobre un valor acumulado (de ahí su nombre) y un elemento extraído de la secuencia. El listado 21.58 muestra un ejemplo de uso.

partial_sum()

Calcula la «suma parcial» para cada elemento de una secuencia y lo almacena sobre un iterador de salida:

Listado 21.76: Ejemplo de uso de `partial_sum()`

```
1 void test_partial_sum() {
2   const int size = 5;
```

```
3   vector<int> values(size);
4   fill(values.begin(), values.end(), 1);
5
6   partial_sum(values.begin(), values.end(), values.begin());
7
8   int expected[size] = {1, 2, 3, 4, 5};
9   assert(equal(values.begin(), values.end(), expected));
10 }
```

`adjacent_difference()`

Calcula las diferencias entre elementos consecutivos de la secuencia de entrada y los escribe sobre el iterador de salida:

Listado 21.77: Ejemplo de uso de `adjacent_difference()`

```
1  void test_adjacent_difference() {
2    int values[] = {1, 3, 0, 10, 15};
3    const int size = sizeof(values) / sizeof(int);
4    int result[size];
5
6    adjacent_difference(values, values+size, result);
7
8    int expected[size] = {1, 2, -3, 10, 5};
9    assert(equal(result, result+size, expected));
10 }
```

21.3.9. Ejemplo: inventario de armas

Veamos un programa concreto que ilustra como sacar partido de las algoritmos genéricos. Se trata del típico inventario de armas habitual en cualquier videojuego tipo «shooter».

Lo primero es definir una clase para describir el comportamiento y atributos de cada arma (clase `Weapon*`). El único atributo es la munición disponible. Tiene otras dos propiedades (accesibles a través de métodos virtuales) que indican la potencia del disparo y la cantidad máxima de munición que permite:

Listado 21.78: Inventario de armas: Clase `Weapon`

```
1  class Weapon {
2    int ammo;
3
4  protected:
5    virtual int power(void) const = 0;
6    virtual int max_ammo(void) const = 0;
7
8  public:
9    Weapon(int ammo=0) : ammo(ammo) { }
10
11   void shoot(void) {
12     if (ammo > 0) ammo--;
13   }
14
15   bool is_empty(void) {
16     return ammo == 0;
17   }
18
19   int get_ammo(void) {
20     return ammo;
```

```
21    }
22
23    void add_ammo(int amount) {
24      ammo = min(ammo + amount, max_ammo());
25    }
26
27    void add_ammo(Weapon* other) {
28      add_ammo(other->ammo);
29    }
30
31    int less_powerful_than(Weapon* other) const {
32      return power() < other->power();
33    }
34
35    bool same_weapon_as(Weapon* other) {
36      return typeid(*this) == typeid(*other);
37    }
38  };
```

Note cómo los métodos `shoot()`, `is_empty()` y `get_ammo()` son autoexplicativos. El método `add_ammo()` está sobrecargado. La primera versión (línea 23) añade al arma la cantidad especificada de balas respetando el límite. Para esto se utiliza el algoritmo `min()`.

El método `less_powerful_than()` compara esta instancia de arma con otra para decidir cuál es la más potente, y por último, el método `same_weapon_as()` indica si el arma es del mismo tipo utilizando RTTI.

El siguiente listado muestra tres especializaciones de la clase `Weapon` que únicamente especializan los métodos privados `power()` y `max_ammo()` para cada uno de los tipos `Pistol`, `Shotgun` y `RPG`.

Listado 21.79: Especializaciones de `Weapon`

```
1  class Pistol : public Weapon {
2    virtual int power(void) const    { return 1; };
3    virtual int max_ammo(void) const { return 50; };
4
5  public:
6    Pistol(int ammo=0) : Weapon(ammo) {}
7  };
8
9  class Shotgun : public Weapon {
10   virtual int power(void) const    { return 10; };
11   virtual int max_ammo(void) const { return 100; };
12
13 public:
14   Shotgun(int ammo=0) : Weapon(ammo) {}
15 };
16
17 class RPG : public Weapon {
18   virtual int power(void) const    { return 100; };
19   virtual int max_ammo(void) const { return 5; };
20
21 public:
22   RPG(int ammo=0) : Weapon(ammo) {}
23 };
```

Veamos por último la clase `Inventory` que representaría la colección de armas que lleva el jugador.

Listado 21.80: Inventario de armas: Clase `Inventory`

```
1  class Inventory : public vector<Weapon*> {
2  public:
```

```
3   typedef typename Inventory::const_iterator WeaponIter;
4   typedef vector<Weapon*> WeaponVector;
5   class WeaponNotFound {};
6   ~Inventory();
7
8   void add(Weapon* weapon) {
9     WeaponIter it =
10      find_if(begin(), end(),
11              bind2nd(mem_fun(&Weapon::same_weapon_as), weapon));
12
13    if (it != end()) {
14      (*it)->add_ammo(weapon);
15      delete weapon;
16      return;
17    }
18
19    push_back(weapon);
20  }
21
22  WeaponVector weapons_with_ammo(void) {
23    WeaponVector retval;
24
25    remove_copy_if(begin(), end(), back_inserter(retval),
26                   mem_fun(&Weapon::is_empty));
27
28    if (retval.begin() == retval.end())
29      throw Inventory::WeaponNotFound();
30
31    return retval;
32  }
33
34  Weapon* more_powerful_weapon(void) {
35    WeaponVector weapons = weapons_with_ammo();
36
37    sort(weapons.begin(), weapons.end(),
38         mem_fun(&Weapon::less_powerful_than));
39
40    return *(weapons.end()-1);
41  }
```

Algunos detalles interesantes de esta clase:

- `Inventory` «es-un» contenedor de punteros a `Weapon`, concretamente un vector (`vector<Weapon*>`) como se puede apreciar en la línea 1.
- La línea 3 define el tipo `WeaponIter` como alias del tipo del iterador para recorrer el contenedor.
- En la línea 4, la clase `WeaponNotFound` se utiliza como excepción en las búsquedas de armas, como veremos a continuación.

El método `add()` se utiliza para añadir un nuevo arma al inventario, pero contempla específicamente el caso —habitual en los *shotters*— en el que coger un arma que ya tiene el jugador implica únicamente coger su munición y desechar el arma. Para ello, utiliza el algoritmo `find_if()` para recorrer el propio contenedor especificando como predicado el método `Weapon::same_weapon_as()`. Nótese el uso de los adaptadores `mem_fun()` (por tratarse de un método) y de `bind2nd()` para pasar a dicho método la instancia del arma a buscar. Si se encuentra un arma del mismo tipo (líneas 12–16) se añade su munición al arma existente usando el iterador devuelto por `find_if()` y se elimina (línea 14). En otro caso se añade la nueva arma al inventario (línea 18).

Por otra parte, el método `more_powerful_weapon()` (líneas 34–41) implementa una funcionalidad también muy habitual en ese tipo de juegos: cambiar al arma más potente disponible. En este contexto, invoca `weapons_with_ammo()` (líneas 22–32) para obtener las armas con munición. Utiliza el algoritmo `remove_copy_if()` para crear un vector de punteros (mediante la función `back_inserter()`), evitando copiar las vacías (línea 26).

Ordena el vector resultante usando `sort()` y utilizando como predicado el método `less_powerful_than()` que vimos antes. Por último, el método retorna un puntero al último arma (línea 40). Nótese que el '`*`' en esa línea es la de-referencia del iterador (que apunta a un puntero).

Para acabar, se muestra el destructor de la clase, que se encarga de liberar los punteros que almacena:

Listado 21.81: Inventario de armas: Destructor

```
1  template<class T>
2  T* deleter(T* x) {
3    delete x;
4    return 0;
5  }
6      sort(weapons.begin(), weapons.end(),
7           mem_fun(&Weapon::less_powerful_than));
8
9      return *(weapons.end()-1);
10   }
```

Aquí se utiliza el functor (`deleter`) con el algoritmo `transform()` para liberar cada puntero. La razón de usar `transform()` en lugar de `for_each()` es eliminar las direcciones de los punteros que dejan de ser válidos en el contenedor. Después se borra todo el contenedor usando su método `clear()`.

21.4. Aspectos avanzados de la STL

En esta sección veremos cómo explotar el potencial de la librería STL más allá del mero uso de sus contenedores y algoritmos.

21.4.1. Eficiencia

La eficiencia es sin duda alguna uno de los objetivos principales de la librería STL. Esto es así hasta el punto de que se obvian muchas comprobaciones que harían su uso más seguro y productivo. El principio de diseño aplicado aquí es:

> Es factible construir *decoradores* que añadan comprobaciones adicionales a la versión eficiente. Sin embargo no es posible construir una versión eficiente a partir de una segura que realiza dichas comprobaciones.

Algunas de estas comprobaciones incluyen la dereferencia de iteradores nulos, invalidados o fuera de los límites del contenedor, como se muestra en el siguiente listado.

Para subsanar esta situación el programador puede optar entre utilizar una implementación que incorpore medidas de seguridad —con la consiguiente reducción de eficiencia— o bien especializar los contenedores en clases propias y controlar específicamente las operaciones susceptibles de ocasionar problemas.

Principio de Pareto

El principio de Pareto también es aplicable a la ejecución de un programa. Estadísticamente el 80 % del tiempo de ejecución de un programa es debido únicamente al 20 % de su código. Eso significa que mejorando ese 20 % se pueden conseguir importantes mejoras. Por ese motivo, la optimización del programa (si se necesita) debería ocurrir únicamente cuando se haya identificado dicho código por medio de herramientas de perfilado y un adecuado análisis de los flujos de ejecución. Preocuparse por optimizar una función lenta que solo se invoca en el arranque de un servidor que se ejecuta durante días es perjudicial. Supone un gasto de recursos y tiempo que probablemente producirá código menos legible y mantenible.

En cualquier caso el programador debería tener muy presente que este tipo de decisiones ad hoc (eliminar comprobaciones) forman parte de la fase de optimización y sólo deberían considerarse cuando se detecten problemas de rendimiento. En general, tal como dice Ken Beck, «La optimización prematura es un lastre». Es costosa (en tiempo y recursos) y produce normalmente código más sucio, difícil de leer y mantener, y por tanto, de inferior calidad.

Listado 21.82: Situaciones no controladas en el uso de iteradores

```
1  void test_lost_iterator() {
2    vector<int>::iterator it;
3    int i = *it;  // probably a SEGFAULT
4  }
5
6  void test_invalidated_iterator() {
7    vector<int> v1;
8    v1.push_back(1);
9    vector<int>::iterator it = v1.begin();
10   v1.clear();
11
12   int i = *it; // probably a SEGFAULT
13 }
14
15 void test_outbound_iterator() {
16   vector<int> v1;
17   vector<int>::iterator it = v1.end();
18
19   int i = *it; // probably a SEGFAULT
20 }
```

Sin embargo, existen otro tipo de decisiones que el programador puede tomar cuando utiliza la STL, que tienen un gran impacto en la eficiencia del resultado y que no afectan en absoluto a la legibilidad y mantenimiento del código. Estas decisiones tienen que ver con la elección del contenedor o algoritmo adecuado para cada problema concreto. Esto requiere conocer con cierto detalle el funcionamiento y diseño de los mismos.

Elegir el contenedor adecuado

A continuación se listan los aspectos más relevantes que se deberían tener en cuenta al elegir un contenedor, considerando las operaciones que se realizarán sobre él:

- Tamaño medio del contenedor.

  En general, la eficiencia –en cuanto a tiempo de acceso– solo es significativa para grandes cantidades de elementos. Para menos de 100 elementos (seguramente muchos más considerando las computadoras o consolas actuales) es muy probable que la diferencia entre un contenedor con tiempo de acceso lineal y uno logarítmico sea imperceptible. Si lo previsible es que el número de elementos sea relativamente pequeño o no se conoce bien a priori, la opción más adecuada es `vector`.

- Inserción de elementos en los dos extremos de la secuencia.

  Si necesita añadir al comienzo con cierta frecuencia (>10 %) debería elegir un contenedor que implemente esta operación de forma eficiente como `deque`.

- Inserción y borrado en posiciones intermedias.

  El contenedor más adecuado en este caso es `list`. Al estar implementado como una lista doblemente enlazada, la operación de inserción o borrado implica poco más que actualizar dos punteros.

- Contenedores ordenados.

  Algunos contenedores, como `set` y `multiset`, aceptan un operador de ordenación en el momento de su instanciación. Después de cualquier operación de inserción o borrado el contenedor quedará ordenado. Esto es órdenes de magnitud más rápido que utilizar un algoritmo de ordenación cuando se necesite ordenarlo.

Otro aspecto a tener en cuenta es la distinción entre contenedores basados en bloques (como `vector`, `deque` o `string`) y los basados en nodos (como `list`, `set`, `map`, etc.). Los contenedores basados en nodos almacenan cada elemento como unidades independientes y se relacionan con los demás a través de punteros. Esto tiene varias implicaciones interesantes:

- Si se obtiene un iterador a un nodo, sigue siendo válido durante toda la vida del elemento. Sin embargo, en los basados en bloque los iteradores pueden quedar invalidados si se realoja el contenedor.
- Ocupan más memoria por cada elemento almacenado, debido a que se requiere información adicional para mantener la estructura: árbol o lista enlazada.

Elegir el algoritmo adecuado

Aunque los algoritmos de STL están diseñados para ser eficientes (el estándar incluso determina la complejidad ciclomática máxima permitida) ciertas operaciones sobre grandes colecciones de elementos pueden implicar tiempos de cómputo muy considerables. Para reducir el número de operaciones a ejecutar es importante considerar los condicionantes específicos de cada problema.

Uno de los detalles más simples a tener en cuenta es la forma en la que se especifica la entrada al algoritmo. En la mayoría de ellos la secuencia queda determinada por el iterador de inicio y el de fin. Lo interesante de esta interfaz es que darle al algoritmo parte del contenedor es tan sencillo como dárselo completo. Se pueden dar innumerables situaciones en las que es perfectamente válido aplicar cualquiera de los algoritmos genéricos que hemos visto a una pequeña parte del contenedor. Copiar, buscar, reemplazar o borrar en los n primeros o últimos elementos puede servir para lograr el objetivo ahorrando muchas operaciones innecesarias.

Otra forma de ahorrar cómputo es utilizar algoritmos que hacen solo parte del trabajo (pero suficiente en muchos casos), en particular los de ordenación y búsqueda como `partial_sort()`, `nth_element()`, `lower_bound()`, etc.

Por ejemplo, una mejora bastante evidente que se puede hacer a nuestro *inventario de armas* (ver listado 21.80) es cambiar el algoritmo `sort()` por `max_element()` en el método `more_powerful_weapon()`.

Listado 21.83: Modificación del inventario de armas

```
1  Weapon* more_powerful_weapon(void) {
2    WeaponVector weapons = weapons_with_ammo();
3
4    return *max_element(weapons.begin(), weapons.end(),
5            mem_fun(&Weapon::less_powerful_than));
6  }
```

Aunque no es previsible que sea un contenedor con muchos elementos, buscar el máximo (que es la verdadera intención del método) es mucho más rápido que ordenar la colección y elegir el último.

Algoritmos versus métodos del contenedor

Utilizar los algoritmos genéricos de STL facilita –obviamente– escribir código (o nuevos algoritmos) que pueden operar sobre cualquier contenedor. Lamentablemente, como no podía ser menos, la generalidad suele ir en detrimento de la eficiencia. El algoritmo genérico desconoce intencionadamente los detalles de implementación de cada contenedor, y eso implica que no puede (ni debe) aprovecharlos para trabajar del modo más eficiente posible. Resumiendo, para el algoritmo genérico es más importante ser genérico que eficiente.

En aquellos casos en los que la eficiencia sea más importante que la generalidad (y eso también hay que pensarlo con calma) puede ser más adecuado utilizar los métodos del contenedor en lugar de sus algoritmos funcionalmente equivalentes. Veamos el siguiente listado:

Listado 21.84: Algoritmo genérico vs. método del contenedor

```
1  void test_algorithm_vs_method(void) {
2    int orig[] = {1, 2, 3, 4, 5};
3    const int SIZE = sizeof(orig) / sizeof(int);
4    vector <int> v1, v2;
5
6    copy(orig, orig + SIZE, back_inserter(v1));
7
8    v2.insert(v2.begin(), orig, orig + SIZE);
9
10   assert(equal(v1.begin(), v1.end(), v2.begin()));
11 }
```

Las líneas 6 y 8 realizan la misma operación: añadir a un `vector` el contenido del array `orig`, creando elementos nuevos (los vectores están vacíos). Sin embargo, la versión con `insert()` (línea 8) es más eficiente que `copy()`, ya que realiza menos copias de los elementos.

Del mismo modo, aunque parece más evidente, utilizar métodos en los que se pueden especificar rangos es siempre más eficiente que usar sus equivalentes en los que sólo se proporciona un elemento (muchos métodos están sobrecargados para soportar ambos casos).

El libro «Effective STL» [63] de Scott Meyers explica muchas otras «reglas» concretas en las que el uso adecuado de STL puede aumentar notablemente la eficiencia del programa.

21.4.2. Semántica de copia

Una cuestión que a menudo confunde a los programadores novatos es la semántica de copia de STL. Significa que los contenedores almacenan copias de los elementos añadidos, y del mismo modo, devuelven copias cuando se extraen. El siguiente listado ilustra este hecho.

Listado 21.85: Semántica de copia de la STL

```
1  class Counter {
2    int value;
3  public:
4    Counter(void) : value(0) {}
5    void inc(void) { ++value; }
6    int get(void)  { return value; }
7  };
8
```

```
 9 void test_copy_semantics(void) {
10   vector<Counter> counters;
11   Counter c1;
12   counters.push_back(c1);
13   Counter c2 = counters[0];
14
15   counters[0].inc();
16
17   assert(c1.get() == 0);
18   assert(counters[0].get() == 1);
19   assert(c2.get() == 0);
20 }
```

Esto tiene graves implicaciones en la eficiencia de las operaciones que se realizan sobre el contenedor. Todos los algoritmos que impliquen añadir, mover y eliminar elementos dentro de la secuencia (la práctica totalidad de ellos) realizan copias, al menos cuando se trata de contenedores basados en bloque.

El siguiente listado es un «decorador» bastante rudimentario para `string` que imprime información cada vez que una instancia es creada, copiada o destruida.

Listado 21.86: Semántica de copia de la STL

```
 1 class String {
 2   string value;
 3   string desc;
 4
 5 public:
 6   String(string init) : value(init), desc(init) {
 7     cout << "created: " << desc << endl;
 8   }
 9   String(const String& other) {
10     value = other.value;
11     desc = "copy of " + other.desc;
12     cout << desc << endl;
13   }
14   ~String() {
15     cout << "destroyed: " << desc << endl;
16   }
17   bool operator<(const String& other) const {
18     return value < other.value;
19   }
20   friend ostream&
21   operator<<(ostream& out, const String& str) {
22     out << str.value;
23     return out;
24   }
25 };
26
27 void test_copy_semantics(void) {
28   vector<String> names;
29   names.push_back(String("foo"));
30   names.push_back(String("bar"));
31   names.push_back(String("buzz"));
32   cout << "-- init ready" << endl;
33
34   sort(names.begin(), names.end());
35   cout << "-- sort complete" << endl;
36   String i1 = names.front();
37   cout << "-- end" << endl;
```

El resultado al ejecutarlo puede resultar sorprendente:

```
created: foo
copy of foo
destroyed: foo
```

```
created: bar
copy of bar
copy of copy of foo
destroyed: copy of foo
destroyed: bar
created: buzz
copy of buzz
copy of copy of copy of foo
copy of copy of bar
destroyed: copy of copy of foo
destroyed: copy of bar
destroyed: buzz
-- init ready
copy of copy of copy of bar
destroyed: copy of copy of copy of bar
copy of copy of buzz
destroyed: copy of copy of buzz
-- sort complete
copy of copy of copy of copy of bar
-- end
destroyed: copy of copy of copy of copy of bar
destroyed: copy of copy of copy of bar
destroyed: copy of copy of buzz
destroyed: copy of copy of copy of foo
```

Como se puede comprobar, las 6 primeras copias corresponden a las inserciones (`push_back()`). El vector reubica todo el contenido cada vez que tiene que ampliar la memoria necesaria, y eso le obliga a copiar en la nueva ubicación los elementos que ya tenía. El algoritmo `sort()` reordena el vector usando solo 2 copias. La asignación implica una copia más. Por último se destruyen los tres objetos que almacena el contenedor y la variable local.

Este ejemplo demuestra la importancia de que nuestras clases dispongan de un constructor de copia correcto y eficiente. Incluso así, muchos programadores prefieren utilizar los contenedores para almacenar punteros en lugar de copias de los objetos, dado que los punteros son simples enteros, su copia es simple, directa y mucho más eficiente. Sin embargo, almacenar punteros es siempre más arriesgado y complica el proceso de limpieza. Si no se tiene cuidado, puede quedar memoria sin liberar, algo difícil de localizar y depurar. Los contenedores no liberan (`delete()`) los punteros que contienen al destruirse. Debe hacerlo el programador explícitamente (ver listado 21.81).

Un punto intermedio entre la eficiencia de almacenar punteros y la seguridad de almacenar copias es utilizar *smart pointers* (aunque **nunca** deben ser `auto_ptr`). Para profundizar en este asunto vea «Implementing Reference Semantics» [51].

21.4.3. Extendiendo la STL

La librería STL está específicamente diseñada para que se pueda extender y adaptar de forma sencilla y eficiente. En esta sección veremos cómo crear o adaptar nuestros propios contenedores, functors y allocators. Ya vimos como crear un algoritmo en la sección 21.3.1.

Creando un contenedor

Dependiendo del modo en que se puede utilizar, los contenedores se clasifican por modelos. A menudo, soportar un modelo implica la existencia de métodos concretos. Los siguientes son los modelos más importantes:

Forward container
Son aquellos que se organizan con un orden bien definido, que no puede cambiar en usos sucesivos. La característica más interesante es que se puede crear más de un iterador válido al mismo tiempo.

Reversible container
Puede ser iterado de principio a fin y viceversa.

Random-access container
Es posible acceder a cualquier elemento del contenedor empleando el mismo tiempo independientemente de su posición.

Front insertion sequence
Permite añadir elementos al comienzo.

Back insertion sequence
Permite añadir elementos al final.

Associative container
Aquellos que permiten acceder a los elementos en función de valores clave en lugar de posiciones.

Cada tipo de contenedor determina qué tipo de iteradores pueden aplicarse para recorrerlo y por tanto qué algoritmos pueden utilizarse con él.

Para ilustrar cuáles son las operaciones que debe soportar un contenedor se incluye a continuación la implementación de `carray`. Se trata de una adaptación (*wrapper*) para utilizar un array C de tamaño constante, ofreciendo la interfaz habitual de un contenedor. En concreto se trata de una modificación de la clase `carray` propuesta inicialmente por Bjarne Stroustrup en su libro «The C++ Programming Language» [93] y que aparece en [51].

Listado 21.87: `carray`: Wrapper STL para un array C

```
template<class T, size_t thesize>
class carray {

private:
  T v[thesize];

public:
  typedef T value_type;
  typedef T* iterator;
  typedef const T* const_iterator;
  typedef T& reference;
  typedef const T& const_reference;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;

  // iteradores
  iterator begin() { return v; }
  const_iterator begin() const { return v; }
  iterator end() { return v+thesize; }
  const_iterator end() const { return v+thesize; }

  // acceso directo
  reference operator[](size_t i) { return v[i]; }
  const_reference operator[](size_t i) const { return v[i]; }

  // size
  size_type size() const { return thesize; }
  size_type max_size() const { return thesize; }

  // conversión a array
```

```
31   T* as_array() { return v; }
32 };
```

El siguiente listado muestra una prueba de su uso. Como los iteradores de `carray` son realmente punteros ordinarios[16], este contenedor soporta los modelos *forward* y *reverse container* además de *random access* ya que también dispone del operador de indexación.

Listado 21.88: `carray`: Ejemplo de uso de `carray`

```
1  void test_carray() {
2    carray<int, 5> array;
3
4    for (unsigned i=0; i<array.size(); ++i)
5      array[i] = i+1;
6
7    reverse(array.begin(), array.end());
8
9    transform(array.begin(), array.end(),
10         array.begin(), negate<int>());
11
12   int expected[] = {-5, -4, -3, -2, -1};
13
14   assert(equal(array.begin(), array.end(), expected));
15 }
```

Functor adaptables

Los adaptadores que incorpora la librería (`ptr_fun()`, `mem_fun()`, etc.) ofrecen suficiente flexibilidad como para aprovechar los algoritmos genéricos utilizando predicados u operadores implementados como métodos o funciones. Aún así, en muchos casos puede ser conveniente escribir functors específicos (ver sección 21.3.3).

Como vimos en la sección 21.3.4 existen adaptadores (`bind1st()`, `not()` o `compose()`, etc.) que necesitan conocer el tipo de retorno o de los argumentos del operador que se le pasa. Estos adaptadores requieren un tipo especial de functor, llamado *functor adaptable*, que contiene las definiciones de esos tipos (como `typedefs`). Ese es el motivo por el que no se puede pasar una función convencional a estos adaptadores. Es necesario usar `ptr_fun()` para «convertir» la función convencional en un *functor adaptable*.

Del mismo modo que los predicados y operadores, STL considera los tipos de functors adaptables correspondientes. Así pues:

- Los Generadores adaptables deberán tener un campo con la definición anidada para su tipo de retorno llamada `result_type`.
- Las Funciones unarias adaptables, además del tipo de retorno, deben especificar además el tipo de su único argumento, con el campo `argument_type`. En el caso de los predicados adaptables se asume que el tipo de retorno es siempre booleano.
- Las Funciones binarias adaptables, además del tipo de retorno, deben especificar el tipo de sus dos argumentos en los campos `first_argument_type` y `second_argument_type`. Del mismo modo, los Predicados binarios no necesitan especificar el tipo de retorno porque se asume que debe ser booleano.

[16] No es extraño encontrar implementaciones de contenedores (como `vector`) perfectamente afines al estándar que utilizan punteros convencionales como iteradores

Veamos la implementación del `ptr_fun()` de g++-4.6 para funciones unarias, que demuestra la utilidad de los functor adaptables:

Listado 21.89: Implementación de `ptr_fun()`

```
1 template<typename _Arg, typename _Result>
2 inline pointer_to_unary_function<_Arg, _Result>
3 ptr_fun(_Result (*__x)(_Arg)) {
4   return pointer_to_unary_function<_Arg, _Result>(__x);
5 }
```

Vemos que `ptr_fun()` es una función-plantilla que se instancia (línea 1) con el tipo del argumento (`_Arg`) y el tipo de retorno (`_Result`). La función devuelve una instancia de `pointer_to_unary_function` (línea 2) instanciada con los mismos tipos. Y el argumento de la función es un puntero a otra función (línea 4) que obviamente devuelve y acepta un parámetro de los tipos indicados en la plantilla. En resumen, `ptr_fun()` es una factoría que crea instancias del functor unario adaptable `pointer_to_unary_function`.

Para facilitar la creación de functor adaptables, STL ofrece plantillas[17] que permiten definir los tipos anidados anteriores para los tipos `unary_function` y `binary_function`. Veamos cómo convertir nuestro functor `not_equal` (ver listado 21.57) en un predicado unario adaptable:

Listado 21.90: Predicado `not_equal()` *adaptable*

```
1  template <typename _Arg>
2  class not_equal : public unary_function<_Arg, bool> {
3    const _Arg _ref;
4
5  public:
6    not_equal(_Arg ref) : _ref(ref) {}
7
8    bool operator()(_Arg value) const {
9      return value != _ref;
10   }
11 };
```

21.4.4. Allocators

Los contenedores ocultan el manejo de la memoria requerida para almacenar los elementos que contienen. Aunque en la gran mayoría de las situaciones el comportamiento por defecto es el más adecuado, pueden darse situaciones en las que el programador necesita más control sobre el modo en que se pide y libera la memoria. Algunos de esos motivos pueden ser:

- Realizar una reserva contigua, reserva perezosa, cacheado, etc.
- Registrar todas las operaciones de petición y liberación de memoria para determinar cuando ocurren y qué parte del programa es la responsable.
- Las características de la arquitectura concreta en la que se ejecuta el programa permiten un manejo más rápido o eficiente de la memoria si se realiza de un modo específico.
- La aplicación permite compartir memoria entre contenedores.

[17] En el fichero de cabecera `<functional>`

- Hacer una inicialización especial de la memoria o alguna operación de limpieza adicional.

Para lograrlo la STL proporciona una nueva abstracción: el *allocator*. Todos los contenedores estándar utilizan por defecto un tipo de allocator concreto y permiten especificar uno alternativo en el momento de su creación, como un parámetro de la plantilla.

Usar un `allocator` alternativo

Como sabemos, todos los contenedores de STL son plantillas que se instancian con el tipo de dato que van a contener. Sin embargo, tienen un segundo parámetro: el `allocator` que debe aplicar. Veamos las primeras líneas de al definición de `vector`.

Listado 21.91: Definición del contenedor `vector`

```
1 template<typename _Tp, typename _Alloc = std::allocator<_Tp> >
2 class vector : protected _Vector_base<_Tp, _Alloc>
3 {
4   typedef typename _Alloc::value_type _Alloc_value_type;
```

Ese parámetro de la plantilla (`_Alloc`) es opcional porque la definición proporciona un valor por defecto (`std::allocator`). El `allocator` también es una plantilla que se instancia con el tipo de elementos del contenedor.

Si se desea utilizar un allocator basta con indicarlo al instanciar el contenedor:

Listado 21.92: Especificando un allocator alternativo

```
1 vector<int, custom_alloc> v;
```

Creando un `allocator`

El allocator es una clase que encapsula las operaciones de petición (a través del método `allocate()`) y liberación (a través del método `deallocate()`) de una cantidad de elementos de un tipo concreto. La signatura de estos métodos es:

Listado 21.93: Métodos básicos del *allocator*

```
1 pointer allocate(size_type n, const void* hint=0);
2 void deallocate(pointer p, size_type n);
```

Crear un allocator no es una tarea sencilla. Lo aconsejable es buscar una librería que proporcione allocators con la funcionalidad deseada, por ejemplo el `pool_alloc` de Boost. Para entender cómo crear un allocator, sin tener que manejar la complejidad que conlleva diseñar y manipular un modelo de memoria especial, se muestra a continuación un wrapper rudimentario para los operadores `new()` y `delete()` estándar. Es una modificación del que propone [51] en la sección 15.4.

Listado 21.94: Un `allocator` básico con `new` y `delete`

```
1 template <class T>
2 class custom_alloc {
```

```
 3 public:
 4   typedef T value_type;
 5   typedef T* pointer;
 6   typedef const T* const_pointer;
 7   typedef T& reference;
 8   typedef const T& const_reference;
 9   typedef size_t size_type;
10   typedef ptrdiff_t difference_type;
11
12
13   template <typename U>
14   struct rebind {
15     typedef custom_alloc<U> other;
16   };
17
18   custom_alloc() {}
19
20   custom_alloc(const custom_alloc&) {}
21
22   template <typename U>
23   custom_alloc(const custom_alloc<U>&) {}
24
25   pointer address(reference value) const {
26     return &value;
27   }
28
29   const_pointer address(const_reference value) const {
30     return &value;
31   }
32
33   size_type max_size() const {
34     return numeric_limits<size_t>::max() / sizeof(T);
35   }
36
37   pointer allocate(size_type n, const void* hint=0) {
38     return (pointer) (::operator new(n * sizeof(T)));
39   }
40
41   void deallocate(pointer p, size_type num) {
42     delete p;
43   }
44
45   void construct(pointer p, const T& value) {
46     new (p) T(value);
47   }
48
49   void destroy(pointer p) {
50     p->~T();
51   }
```

Las líneas 4 a 15 definen una serie de tipos anidados que todo allocator debe tener:

`value_type`
El tipo del dato del objeto que almacena.

`reference` y `const_reference`
El tipo de las referencia a los objetos.

`pointer` y `const_pointer`
El tipo de los punteros a los objetos.

`size_type`
El tipo que representa los tamaños (en bytes) de los objetos.

`difference_type`
El tipo que representa la diferencia entre dos objetos.

Las líneas 18 a 23 contienen el constructor por defecto, el constructor de copia y un constructor que acepta una instancia del mismo allocator para otro tipo. Todos ellos están vacíos porque este allocator no tiene estado.

El método polimórfico `address()` (líneas 25 a 31) devuelve la dirección del objeto. El método `max_size()` devuelve el mayor valor que se puede almacenar para el tipo concreto.

Por último, los métodos `allocate()` y `deallocate()` sirven para pedir y liberar memoria para el objeto. Los métodos `construct()` y `desctroy()` construyen y destruyen los objetos.

21.4.5. Novedades de la STL en C++11

El nuevo estándar C++11 viene con interesantes novedades a la librería STL. Esta sección aborda las más importantes. Veremos cómo el nuevo estándar mejora la eficiencia y simplifica algunos aspectos en el uso de la librería.

La función `bind()`

En realidad `bind()` es una plantilla. Se utiliza para crear un functor a partir de una función, método o functor y un especificación de argumentos. `bind()` substituye a los adaptadores `bind1st()` y `bind2nd()`. Permite especificar la posición en la que se desea pasar el argumento adicional indicado en el constructor y el de la invocación al functor resultante. Veamos un ejemplo de cómo `bind()` puede substituir a `bind2nd()` en el listado 21.63.

Listado 21.95: Ejemplo de `find_if()` usando `bind()`

```
1  void test_find_if_cpp11() {
2    using namespace std::placeholders;
3
4    const int size = 5;
5    const int value = 1;
6    int numbers[] = {2, 7, 12, 9, 4};
7
8    assert(find_if(numbers, numbers + size,
9                   bind(greater<int>(), _1, 6)) == numbers+1);
```

El argumento especial `_1` es un *placeholder* (ver línea 18) que representa al primer argumento posicional que recibirá el functor generado. Es decir, si asumimos que la invocación a `bind()` crea un hipotético `bool greater_than_6(int n)`, la variable `_1` representa al parámetro `n`.

Con esta nomenclatura tan simple es posible adaptar una llamada a función o método a otra aunque los parámetros estén dispuestos en un orden diferente o pudiendo añadir u omitir algunos de ellos.

También substituye a los adaptadores `ptr_fun()`, `mem_fun()` y, finalmente, `mem_fun_ref()` dado que acepta cualquier entidad *invocable* y funciona tanto con punteros como con referencias. Veamos la versión con `bind()` del listado 21.60:

Listado 21.96: Uso de `bind()` como reemplazo del adaptador `mem_fun_ref()`

```
1  void test_bind_replaces_mem_fun_ref() {
2    using namespace std::placeholders;
3
4    vector<Enemy> enemies(2);
```

```
5
6   assert(count_if(enemies.begin(), enemies.end(),
7           bind(&Enemy::is_alive, _1)) == 2);
8 }
```

Pero `bind()` hace mucho más. Permite construir funciones *parcialmente especificadas*. Una función parcialmente especificada es una utilidad típica de los lenguajes funcionales. Como su nombre indica, es una forma de fijar el valor de uno o más parámetros de una función (o algo que se comporta como tal). El resultado es una nueva función (un functor realmente) con menos parámetros. Veamos un ejemplo (listado 21.97). Tenemos una función que imprime un texto en un color determinado indicado por parámetro. Utilizando `bind()` vamos a crear otra función que, a partiendo de la primera, permite escribir texto en verde para imprimir un mensaje que informa de un comportamiento correcto (como en un *logger*).

`bind()`

Es el adaptador de funciones definitivo. Substituye a todos los adaptadores que ofrecía la STL antes del estándar C++11.

Listado 21.97: Una función parcialmente especificada

```
1 enum Color{black, red, green};
2
3 void print_color(string text, Color color) {
4   cout << text << " color:" << color << endl;
5 }
6
7 void test_bind() {
8   auto print_ok = bind(print_color, _1, green);
9   print_ok("Success");
10 }
```

Las funciones parcialmente especificadas resultan muy útiles cuando se requieren manejadores de eventos. Normalmente éstos deben cumplir un prototipo concreto y ahí es donde `bind()` simplifica las cosas. En el siguiente ejemplo vemos un mapa de manejadores para eventos de teclado en una aplicación Ogre.

Listado 21.98: Creando un mapa de manejadores gracias a `bind()`

```
1  class WindowEventListener: public Ogre::WindowEventListener,
2                 public OIS::KeyListener,
3                 public OIS::MouseListener {
4    OIS::InputManager* inputManager;
5    OIS::Keyboard* keyboard;
6    OIS::Mouse* mouse;
7
8    typedef vector<OIS::KeyCode> KeyCodes;
9    map<KeyCodes, function<void()>> triggers_;
10
11   // ...
12
13 public:
14   void add_hook(KeyCodes keystroke, function<void()> callback) {
15     triggers_[keystroke] = callback;
16   }
17
18   // ...
19 };
20
21 int main() {
22   // ...
23
24   WindowEventListener listener(game.window);
25   Car car(chassis, wheels);
26
27   listener.add_hook({OIS::KC_W, OIS::KC_D},
28                     bind(&Car::forward_right, &car));
29   listener.add_hook({OIS::KC_W, OIS::KC_A},
```

```
30                      bind(&Car::forward_left, &car));
31    listener.add_hook({OIS::KC_S, OIS::KC_D},
32                      bind(&Car::backward_right, &car));
33    listener.add_hook({OIS::KC_S, OIS::KC_A},
34                      bind(&Car::backward_left, &car));
35    listener.add_hook({OIS::KC_W},
36                      bind(&Car::forward, &car));
37    listener.add_hook({OIS::KC_S},
38                      bind(&Car::backward, &car));
39    listener.add_hook({OIS::KC_ESCAPE},
40                      bind(&WindowEventListener::shutdown, &listener)
                            );
41
42    // ...
```

Fíjese en el atributo `triggers_` (línea 9), el cual consiste en un mapa que relaciona un `vector` de `KeyCode` (una combinación de teclas) con un manejador (tipo `function<void()>`). Esto permite cambiar las asignaciones de teclas según las preferencias del usuario, leyéndolas por ejemplo de un fichero de configuración, y sería muy sencillo también cambiarlas con la aplicación en marcha si fuese necesario.

Contenedores

Con el nuevo estándar, todos los tipos se pueden inicializar de forma equivalente a como se hacía con los arrays, es decir, escribiendo los datos entre llaves:

Listado 21.99: Inicialización uniforme: un array C y un `vector`

```
1 int numbers[] = {1, 2, 3, 4};
2 std::vector<int> numbers_vector = {1, 2, 3, 4};
```

Esto es gracias a un tipo llamado `initializer_list` que se puede usar para crear un constructor alternativo. Pero también se puede utilizar en otros métodos como `insert()` o `assign()`:

Listado 21.100: Usando `initializer_list` con `insert()`

```
1 numbers_vector.insert(numbers_vector.end(), {5, 6, 7});
```

Puedes ver más detalles y ejemplos sobre *inicialización uniforme* en la sección 21.5.2.

Se ha añadido el contenedor `array`. Es de tamaño fijo y tan eficiente como un array C nativo, y por supuesto, con una interfaz estilo STL similar a `vector` (ver listado 21.101).

Listado 21.101: El nuevo contenedor `array`

```
1 std::array<int, 4> numbers = {10, 20, 30, 40};
2
3 std::cout << numbers.size() << std::endl;
4 for (auto n: numbers)
5   std::cout << n << " ";
```

Se ha añadido el contenedor `forward_list`. Se trata de una lista enlazada sencilla, que únicamente soporta iteración hacia adelante, no permite acceder al final, ni siquiera dispone del método `size()`.

Otra novedad interesante relacionada con la eficiencia es la aparición de los métodos `emplace()` y `emplace_back()` en varios contenedores. `emplace_back()` construye un objeto directamente en el contenedor, sin crear un objeto temporal, algo muy conveniente cuando nuestros objetos son complejos. Ejecuta el ejemplo del listado 21.102 para comprobar el efecto.

Listado 21.102: Uso de `emplace()` para inserción eficiente

```
 1 struct A {
 2   A(int) {
 3     cout << "ctor" << endl;
 4   }
 5   A(const A& other) {
 6     cout << "copy-ctor" << endl;
 7   }
 8   A(A&& other) {
 9     cout << "move-ctor" << endl;
10   }
11 };
12
13 int main() {
14   std::vector<A> v;
15
16   v.emplace_back(1);
17   cout << "---" << endl;
18   v.push_back(A(1));
19 }
```

Iteradores

El nuevo estándar ha incorporado métodos que retornan iteradores constantes en todos los contenedores: `cbegin()`, `cend()` y también iteradores inversos contantes: `crbegin()` y `crend()`. Además están disponibles versiones de `begin()` y `end()` como funciones. De ese modo es posible escribir código genérico que funciona con una referencia a cualquier contenedor, incluyendo arrays C.

Algoritmos

Se han incorporado varios algoritmos nuevos. Aquí se incluye un resumen de algunos de ellos:

`all_of()` Devuelve cierto sólo si el predicado es cierto para todos los elementos de la secuencia.

`any_of()` Devuelve cierto si el predicado es cierto para al menos uno de los elementos de la secuencia.

`none_of()` Devuelve cierto sólo si el predicado es falso para todos los elementos de la secuencia.

`iota()` Asigna valores crecientes a los elementos de una secuencia.

`minmax()` Devuelve los valores mínima y máximo de una secuencia.

`is_sorted()` Devuelve cierto si la secuencia está ordenada.

21.5. C++11: Novedades del nuevo estándar

El 12 de Agosto de 2011 la Organización Internacional de Estándares (ISO) aprobó el nuevo estándar de C++, anteriormente conocido como C++0x. Además de añadir funcionalidades nuevas al lenguaje, C++11 también amplía la STL, incluyendo en la misma casi todas las plantillas y clases ya presentes en el TR1.

C++11 es compatible hacia atrás con C++98 (también con la corrección de 2003) y con C. Aparte de esta, las cualidades que se han pretendido conseguir con el nuevo estándar incluyen la mejora de rendimiento, una programación más evolucionada y su accesibilidad para los programadores no-expertos sin privar al lenguaje de su potencia habitual.

En los siguientes apartados se introducirán algunas de las nuevas características que añade el estándar.

21.5.1. Compilando con `g++` y `clang`

GCC y `Clang` son los dos compiladores que dan soporte a mayor número de características del nuevo estándar. En el momento de escribir esta documentación, la última versión estable de GCC es la 4.8.2, con la que se puede disfrutar de todas las novedades del núcleo del lenguaje y muchas de la biblioteca estándar.

Para compilar un programa de C++ usando el nuevo estándar hay que utilizar la opción `-std=c++11` al compilar. Por ejemplo:

```
g++ -o main main.cc -std=c++11
```

Normalmente, si no se utiliza esta opción, GCC compilará usando el estándar C++03. `Clang` se usa exactamente de la misma forma.

Si se usa la librería estándar hay que linkarla (con g++ se puede omitir pues lo hace de forma automática):

```
clang -o main main.cc -std=c++11 -lstdc++
```

21.5.2. Cambios en el núcleo del lenguaje

Expresiones constantes

Un compilador de C++ es capaz de optimizar ciertas expresiones que serán siempre constantes, por ejemplo:

```
1   int a = 1 + 2;
2   cout << 3.2 - 4.5 << endl;
3
4   int miArray[4 * 2];
```

En este código, el compilador sustituirá las expresiones anteriores por su valor en tiempo de compilación. De este modo, en cualquier buen compilador, el código anterior no generará ninguna suma, resta o producto. Sin embargo, C++03 no permite utilizar funciones que devuelvan constantes (por ejemplo `return 5;`).

C++11 introduce la palabra reservada `constexpr` para brindar la posibilidad de utilizar funciones como expresiones constantes. Anteriormente no era posible puesto que el compilador no tenía ninguna forma de saber que podía aplicar esta optimización. De este modo, es posible escribir algo como lo siguiente:

```
1 constexpr int siete(){ return 7; }
2
3 void miFunc(){
4   char cadena[siete() + 3];
5   cadena[0]='\0';
6   // ...
7 }
```

Un función se podrá declarar como `constexpr` siempre que no devuelva `void` y que termine del modo `return <expresión>`. Dicha expresión tendrá que ser constante una vez que se sustituyan todas las variables y si llama a otras funciones tendrán que estar definidas como `constexpr`.

Es posible declarar variables utilizando `constexpr` que equivale al uso de `const`.

```
1   constexpr int saludJefeNivel = 1337;
2
3   const int saludJefeFinal    = 31337;
4   const int saludJefeEspecial = 3110 + siete();
```

La introducción de esta característica en muy útil con las plantillas. El siguiente código se evaluará en tiempo de compilación y no en tiempo de ejecución, sustituyendo la llamada por el valor devuelto.

```
1 template<typename T> constexpr T max(T a, T b)
2 {
3   return a < b ? b : a;
4 }
```

Inicializador de listas

Antes de la entrada del nuevo estándar, la inicialización de los contenedores de la STL era posible utilizando una zona de memoria con una secuencia de elementos del tipo instanciado. Normalmente se utiliza un `array` para llevar esto a cabo.

Definiendo una estructura del siguiente modo

```
1 struct miStruct {
2   int   a;
3   float b;
4 };
```

se podría inicializar un vector como sigue (también se incluyen ejemplos con enteros).

```
1   miStruct mS[] = { {0, 1.0}, {0, 0.0}, {0, 0.0} };
2   vector<miStruct> mVS(mS, mS + sizeof(mS)/sizeof(miStruct));
3
4   int mA[] = {1, 1, 1, 2, 3, 4, 1};
5   vector<int> mVI(mA, mA + sizeof(mA)/sizeof(int));
6
7   int mB[] = {1, 2, 3, 4};
8   set<int> mC(mB, mB + sizeof(mB)/sizeof(int));
```

A partir de ahora, es posible utilizar lo que se conoce como el inicializador de listas, que permite realizar inicializaciones de manera mucho más sencilla.

```
1   vector<miStruct> miVS {{0, 0.0}, {0, 0.0}, {0, 0.0}};
2   vector<int>      miVI {1, 1, 1, 2, 3, 4, 1};
3   set<int>         miC  {0, 4, 5, 9};
```

Esto es posible gracias al uso de usa nueva sintaxis y del contenedor `std::initializer_list`.

Si se utiliza como parámetro en el constructor de una clase

Listado 21.103: Clase que utiliza un inicializador de listas

```
1 class LODDistancias {
2 public:
3     LODDistancias(std::initializer_list<int> entrada) :
4         distancias(entrada) {}
5 private:
6   vector<int> distancias;
7 };
```

es posible hacer uso de las llaves para inicializarla:

```
1   LODDistancias lodD {90, 21, 32, 32, 35, 45};
```

Hay que tener en cuenta que este tipo de contenedores se utilizan en tiempo de compilación, que sólo pueden ser construidos estáticamente por el compilador y que no podrán ser modificados en tiempo de ejecución. Aun así, como son un tipo, pueden ser utilizados en cualquier tipo de funciones.

También se pueden utilizar las llaves junto con el operador =, para inicializar o para asignar nuevos valores.

```
1   vector<miStruct> miVS2 = {{0, 0.0}, {0, 0.0}, {0, 0.0}};
2   vector<int>      miVI2 = {1, 1, 1, 2, 3, 4, 1};
3   set<int>         miC2  = {0, 4, 5, 9};
4
5   miVS2 = {{9, 1.2}};
```


Inicialización uniforme

En C++03 no existe una forma uniforme de inicializar los objetos. En el aparatado anterior, en la parte compatible con el antiguo estándar, se ha utilizado la inicialización de un *array* utilizando . Esto es posible ya que esa estructura es un agregado[18], ya que sólo este tipo de objetos y los *arrays* pueden ser inicializados de esta manera.

Con la aparición de C++11, es posible utilizar las llaves para inicializar cualquier clase o estructura. Por ejemplo, supongamos una clase para representar un vector de tres dimensiones.

[18]Un agregado (*aggregate*) es una clase o estructura que no tiene destructor definido por el usuario ni operador de asignación. Tampoco tendrán miembros privados o protegidos que no sean estáticos, ni una clase base o funciones virtuales.

```
1 class Vector3D {
2 public:
3   Vector3D(float x, float y, float z):
4     _x(x), _y(y), _z(z) {}
5
6 private:
7   float _x;
8   float _y;
9   float _z;
10
11  friend Vector3D normalize(const Vector3D& v);
12 };
```

Es posible iniciar un objeto de tipo `Vector3D` de las dos formas siguientes.

```
1   Vector3D p{0.0, 1.1, -3.4};
2
3   Vector3D p1(1.8, 1.4, 2.3);
```

La primera utiliza la nueva inicialización uniforme, la segunda la clásica, invocando el constructor de forma explícita.

En C++11 también es posible utilizar esta inicialización para construir de manera implícita objetos que son devueltos por una función. El compilador utilizará el valor de retorno del prototipo de la función y lo usará junto con los valores proporcionados para construir y devolver un objeto de dicho tipo.

```
1 Vector3D normalize(const Vector3D& v){
2   float len = sqrt(v._x*v._x + v._y*v._y + v._z*v._z);
3
4   return {v._x/len, v._y/len, v._z/len};
5 }
```

Esta notación no sustituye a la anterior. Cabe destacar que cuando se utiliza esta sintaxis para inicializar un objeto, el constructor que acepta una lista de inicialización como las presentadas anteriormente tendrá prioridad sobre otros. Debido a esto, algunas veces será necesario invocar directamente al constructor adecuado con la notación antigua.

Esta forma de devolver objetos es compatible con RVO (Return Value Optimization), que se verá en optimizaciones. Con lo cual una llamada como la siguiente generará código óptimo y seguramente sin ninguna copia.

```
1   Vector3D p2 = normalize(p);
```

Inferencia de tipos

Hasta ahora cada vez que se declaraba una variable en C++ había que especificar de qué tipo era de manera explícita. En C++11 existe la inferencia de tipos. Usando la palabra reservada `auto` en una inicialización en vez del tipo, el compilador deducirá el mismo de manera automática.

En el ejemplo siguiente se ve cómo funciona esta característica, tanto para tipos básicos como para la clase definida en el apartado anterior.

```
1   auto vidaJefe  = 500;
2   auto precision = 1.00001;
3
```

```
4   Vector3D v(3.0, 2.1, 4.0);
5   auto v2 = normalize(v);
```

Esta nueva característica es especialmente adecuada para simplificar algunas declaraciones complejas. A continuación se muestra la diferencia en la declaración del iterador al recorrer un contenedor.

```
1   for (vector<double>::iterator it = dist.begin();
2        it != dist.end(); ++it)
3     cout << *it << endl ;
4
5   for (auto it = dist.begin(); it != dist.end(); ++it)
6     cout << *it << endl ;
```

Existe otra palabra reservada que se usa de forma similar a `sizeof()`, pero que devuelve el tipo de una variable. Esta palabra es `decltype` y se puede usar para extraer el tipo de una variable y usarlo para la declaración de otra.

```
1   decltype(v2) otro_vector3d = {4.1, 3.0, 1.1};
```

Bucle `for` basado en rangos

En C++11 se introduce una característica muy útil para recorrer listas de elementos, ya sean *arrays*, lista de inicialización o contenedores con las operaciones `begin()` y `end()`.

```
1   int records[4] = {900, 899, 39, 3};
2   for (int& i: records)
3     cout << i << endl;
4
5   list<float> punteria = {20.0, 10.9};
6   for (float& f: punteria)
7     cout << f << endl;
```

En el ejemplo anterior se utiliza una referencia para evitar la copia y la penalización de rendimiento.

Funciones Lambda

Las funciones lambda son simplemente funciones anónimas. La sintaxis para declarar este tipo de funciones es especial y es posible no declarar el tipo devuelto de manera explícita sino que está definido de forma implícita mediante la construcción `decltype(<expresión_devuelta>)`. Las dos formas posible de declarar y definir estas funciones son las siguientes.

```
[captura](parámetros)->tipo_de_retorno{cuerpo}

[captura](parámetros){cuerpo}
```

La primera hace explícito el tipo que se devuelve. De esto modo, las funciones que se muestran a continuación son equivalentes.

```
1   [](int p1, int p2)->int{ return p1+p2; };
2   [](int p1, int p2){ return p1+p2; };
```

Las variables que se utilizan dentro de estas funciones pueden ser capturadas para utilizarse en el exterior. Se pueden capturar por valor o por referencia, dependiendo de la sintaxis dentro de `[]`. Si se utiliza por ejemplo `[p1, &p2]`, `p1` será capturada por valor y `p2` por referencia. Si se usa `[=,&p1]`, todas las variables serán capturadas por valor (al usar =) excepto `p1` que será capturada por referencia. Si se utiliza `[&,p2]`, todas se capturarán por referencia (usando `&`), excepto `p2`.

En el siguiente ejemplo, se utiliza una función lambda para sumar la puntaciones de todos los jugadores, que han sido previamente almacenadas en una lista. Se muestran tres formas de hacerlo.

```
1   list<int> puntos = {330, 300, 200, 3892, 1222};
2   int suma = 0;
3
4   // 1)
5   auto f = [&suma](int& i){suma+=i;};
6   for (int& i: puntos)
7     f(i);
8   // 2)
9   for_each(puntos.begin(), puntos.end(),
10            [&suma](int& i){suma+=i;} );
11  // 3)
12  for_each(puntos.begin(), puntos.end(), f);
```

Declaración alternativa de funciones

C++11 introduce una nueva forma de declarar funciones. Su utilidad es permitir declarar los tipos de retorno de funciones *templatizadas* donde éste no se puede averiguar a priori.

En el ejemplo siguiente se define una clase y se declaran dos funciones *templatizadas*.

```
1  class K {
2  public:
3    int operator*(const K& k) const {return 2;}
4  };
5
6  template <typename T>
7  T pow2Bad(const T& t){return t*t;}
8
9  template <typename T>
10 auto pow2(const T& t)->decltype(t*t){return t*t;}
```

La primera función no compilará si el tipo que se devuelve al ejecutar la operación es diferente al tipo para el que se invoca. La segunda sí lo hará.

```
1   K kObj;
2   cout << pow2Bad(kObj) << endl; // <- no compila
3   cout << pow2(kObj)    << endl;
```

También se puede usar estar nueva sintaxis para funciones no *templatizadas*.

```
1   auto getHours()->int{ return _hours;}
```

Mejora en la construcción de objetos: delegación

En C++03 es imposible invocar a un constructor desde otro constructor del mismo objeto. En C++11 sí es posible.

```
1  class playerInfo {
2  public:
3    playerInfo(const string& name) :
4      _name(name) {}
5
6    playerInfo() : playerInfo("default") {}
7
8  private:
9    string _name;
10 };
```

Sobrescritura explícita y declaración final

En C++11 es posible utilizar dos descriptores para añadir funcionalidad e información para el compilador a la declaración de los métodos de una clase.

El descriptor `override` proporciona una forma de expresar que el método que se está declarando sobrescribe a otro de una clase base. Esto es útil para expresar explícitamente las intenciones y facilitar la detección de fallos en tiempos de compilación. Así, si se declara un método como usando `override` y no existe uno con el mismo prototipo que éste en una base clase, el compilador mostrará un error.

Listado 21.104: Uso de `final` y `override`

```
1  class Base {
2  public:
3    virtual int  getX(){return _x;}
4    virtual bool isValid() { return true; }
5  private:
6    int _x;
7  };
8
9  class Derivada : public Base {
10 public:
11   //Ok, compila.
12   virtual int getX() override {
13     return _anotherX;
14   }
15   //Fallo al compilar
16   virtual int getX(int a) override {
17     return _anotherX;
18   };
19   bool isValid() final { return false; }
20 private:
21   int _anotherX;
22 };
23
24 class MasDerivada : public Derivada {
25 public:
```

En el ejemplo anterior también se muestra (líneas 4-6 y 22) el uso de `final`. Cuando se utiliza en la declaración de un método, indica que ninguna clase que herede de ésta podrá sobrescribirlo.

Puntero *null*

Se introduce también un nuevo valor sólo asignable a punteros: `nullptr`. Este valor no se puede asignar a ningún otro tipo. En C++03, se usaba el valor 0 para los punteros *null*, de este modo, se podía asignar el valor de estos punteros a un entero o a un booleano. Con `nullptr` esto ya no es posible, ayudando a prevenir errores y a sobrescribir funciones.

```
1   int* c = nullptr;
```

Cabe destacar que es un tipo compatible con los booleanos, pero que no es compatible con los enteros.

```
1   bool isNull = nullptr;
2   int  zero   = nullptr; // <- Error
```

Enumeraciones fuertemente tipadas

En las enumeraciones de C++03 no se podía distinguir el tipo de entero utilizado para las mismas. En C++11 sí, y además se brinda la posibilidad de usar una visibilidad más restrictiva, para agrupar la enumeraciones sin tener que anidarlas dentro de clases.

```
1 enum TipoPortal :unsigned char {
2   NORMAL,
3   MIRROR
4 };
5
6 enum class TipoArma : unsigned short {
7   BLANCA,
8   EXPLOSIVA
9 };
```

Para utilizarlo, se hará igual que en C++03, excepto en el segundo caso.

```
1   TipoPortal ptype = NORMAL;
2   TipoArma atype   = TipoArma::BLANCA;
```

Además de esto, ahora se permite la declaración anticipada (*forward declaration*) de enumeraciones.

Alias de plantillas

Ya que `typedef` no se puede utilizar con plantillas, C++11 incluye una forma de crear alias para las mismas. Se basa en utilizar `using`.

```
1 template<typename T, typename M>
2 class miTipo;
3
4 template<typename N>
5 using miTipo2 = miTipo<N,N>;
```

También se puede utilizar la nueva sintaxis para realizar las definiciones de tipo que se hacían con `typedef`.

```
1   typedef unsigned int uint;
2   using uint = unsigned int;
```

Uniones sin restricciones

Ahora se permite la creación de uniones con la participación de objetos no triviales en las mismas. El siguiente fragmento de código sólo compilará usando el estándar C++11.

```
1  class Vector3D {
2  public:
3    Vector3D(float x, float y, float z) {}
4  };
5
6  union miUnion {
7    int      a;
8    float    b;
9    Vector3D v;
10 };
```

Nuevos literales de cadenas

C++03 no soportaba ningún tipo de codificación Unicode. Sólo se podían utilizar dos tipos de literales: los que estaban entrecomillados (`"hola"`, que se convertían en *arrays* de `const char`, y los entrecomillados con una `L` delante (`L"hola"`), que se transformarán en *arrays* de `const wchar_t`.

Se introduce tres nuevos tipos de literales, para UTF-8, UTF-16 y UTF-32, que serán *arrays* de `const char`, `const char16_t` y `const char32_t` respectivamente.

```
1   const char     cad1[] = u8"Cadena UTF-8";
2   const char16_t cad2[] =  u"Cadena UTF-16";
3   const char32_t cad3[] =  U"Cadena UTF-32";
```

También se permite la construcción de cadenas *raw*, que no interpretarán los caracteres de escape (_), ni las propias comillas (`"`). Para definir este tipo de cadenas se usa `R"(literal)'"`. También es posible usar cadenas *raw* con la modificación Unicode.

```
1   string raw(R"(Cadena "RAW" \n%d')");
2
3   const char16_t rcad2[] = uR"(Cadena UTF-16 RAW\n)";
4   const char32_t rcad3[] = UR"(Cadena UTF-32 RAW%d)";
```

Literales creados a medida

C++11 brinda al programador con la capacidad de crear nuevos tipos de literales. Anteriormente los literales estaban preestablecidos, por ejemplo `9` es un literal entero, `9.0` uno *double*, y `9.0f` uno de tipo *float*.

A partir de ahora se pueden crear nuevos literales usando sufijos. Los sufijos podrán ir detrás de números (los que puedan ser representados por `unsigned long long` o `long double`) o detrás de literales de cadena. Estos sufijos corresponden a funciones con el prototipo `retval operator"" _sufijo ( unsigned long long ).`

```
1 double operator"" _d (unsigned long long i) {
2   return (double) i;
3 }
```

La función anterior define el sufijo `_d`, que podrá ser usado para crear un `double` usando un número natural como literal.

```
1   auto d = 30_d;
```

Un ejemplo un poco más complejo del uso de este operador se expone a continuación. Sea la siguiente una clase que podría representar un vector de tres dimensiones, incluyendo la operación de suma.

```
1  class Vector3D {
2  public:
3    Vector3D() :
4      x_(0), y_(0), z_(0) {} ;
5
6    Vector3D(float x, float y, float z) :
7      x_(x), y_(y), z_(z) {} ;
8
9    Vector3D operator+(const Vector3D& v) {
10     return Vector3D(x_ + v.x_,
11                     y_ + v.y_,
12                     z_ + v.z_  );
13   }
14
15 private:
16   float x_;
17   float y_;
18   float z_;
19
20   friend Vector3D operator"" _vx(long double x);
21   friend Vector3D operator"" _vy(long double y);
22   friend Vector3D operator"" _vz(long double z);
23 };
```

Se podrían definir los siguientes literales de usuario, por ejemplo para construir vectores ortogonales.

```
1  Vector3D operator"" _vx(long double x) {
2    return Vector3D(x, 0, 0);
3  }
4
5  Vector3D operator"" _vy(long double y) {
6    return Vector3D(0, y, 0);
7  }
8
9  Vector3D operator"" _vz(long double z) {
10   return Vector3D(0, 0, z);
11 }
```

Como se definió la suma, se podría crear un vector con la misma.

```
1   auto v = 1.0_vx + 3.0_vy + 8.1_vz;
```

Para utilizar los sufijos con los literales de cadenas, se muestra el siguiente ejemplo, que representa un jugador, con un nombre.

```
1  class Player {
2  public:
3    Player(string name):
4      name_(name) {}
5
6  private:
7    string name_;
8  };
9
10 Player operator"" _player(const char* name, size_t nChars) {
11   return Player(name);
12 };
```

Se podrá entonces crear un jugador como sigue.

```
1   auto p = "bRue"_player;
```

Aserciones estáticas

Algo muy útil que ya incluía Boost es una aserción estática. Este tipo de aserciones se comprobarán en tiempo de compilación, y será el mismo compilador el que avise de la situación no deseada.

En C++11 se puede usar `static_assert (cont-expr, error-message)` para utilizar estas aserciones en cualquier punto del código.

```
1  template <typename T>
2  bool equal(T a, T b, T epsilon) {
3
4    static_assert( sizeof(T) >= 8, "4 bytes como poco" );
5
6    return (a > b - epsilon || a < b + epsilon);
7  }
8
9  int main(int argc, char *argv[])
10 {
11   equal(8.0,  8.0000001,  0.00001);  // OK (double 8 bytes)
12   equal(8.0f, 8.0000001f, 0.00001f); // Error!!
13
14   return 0;
15 }
```

La salida de la compilación será la siguiente:

```
$ g++ -o statica statica.cc -std=c++0x

statica.cc: In instantiation of 'bool equal(T, T, T)
[with T = float]':
statica.cc:17:35:   required from here
statica.cc:9:3: error: static assertion failed:
4 bytes como poco
```

Eliminación y selección por defecto explícita de funciones

En C++11 es posible prohibir el uso de las funciones de una clase, incluyendo los constructores, haciendo uso de la palabra reservada `delete`. Esto es muy útil para evitar que alguien use un constructor no deseado (en C++03 se declaraba como privado para obtener el mismo resultado).

```
1 class NoCopiable {
2 public:
3   NoCopiable(){}
4   NoCopiable(const NoCopiable&) = delete;
5   NoCopiable& operator=(const NoCopiable&) = delete;
6 };
```

También es útil para evitar llamadas implícitas a funciones. En C++03 esto es posible para los constructores, utilizando la palabra reservada `explicit`. En C++11, se pueden evitar la invocación no deseada de funciones usando `delete`.

En el ejemplo siguiente, si no se declara la función que acepta un entero, si se realizase una llamada de tipo `setA(3)`, se realizaría una conversión implícita desde un entero a un *double*. Este tipo de comportamientos no siempre es deseable y puede provocar sorpresas, sobre todo con tipos no-básicos.

```
1 class Ex {
2 public:
3   explicit Ex(double a) :
4     a_(a) {}
5
6   Ex() = default;
7
8   void setA(double a) {
9     a_ = a;
10  }
```

En el mismo ejemplo se usa `default` con uno de los constructores, lo que pide de forma explícita al compilador que él cree uno por defecto.

Constructores de movimiento

Se introduce el concepto de constructor de movimiento, en contraste con el aun necesario constructor de copia. Mientras que es este último se usa para determinar la forma en la que se copian los objetos, el de movimiento determina qué significa mover las propiedades de un objeto a otro (no son dos objetos independientes).

Aunque sea de forma transparente al programador, el compilador genera variables temporales para realizar determinadas operaciones. El constructor de movimiento es una forma de evitar este tipo de variables intermedias (de copia) y así poder optimizar determinadas operaciones (asignaciones normalmente).

Se introduce también el concepto de referencias-*rvalue* (`&&`). Ya que en esta sección se introducen muchas características, esta en concreto sólo se va a mencionar por encima, puesto que profundizar en ella podría llevar tanto como para el resto juntas. Como resumen, decir sobre ellas que son referencias especiales que apuntan a variables sin una dirección de memoria (variables temporales).

El constructor de movimiento se declara como el de copia, pero el parámetro de entrada usa `&&`. Lo mismo se aplica para la versión análoga del operador de asignación.

Es importante recalcar que, cuando se programa un constructor de movimiento, hay que lidiar con el destructor del objeto temporal, puesto que se ejecutará cuando este objeto salga de ámbito. Normalmente esto implica tener la precaución de evitar llamar a un `delete` con un puntero no nulo que apunta a una dirección que ya ha sido liberada. Para ello, al mover el objeto temporal, se tendrá que evitar que se libere la memoria del puntero que se ha movido, asignándole el valor nulo. De esto modo, cuando el objeto temporal salga de ámbito y se ejecute su destructor, `delete` no actuará sobre su puntero.

Listado 21.105: Ejemplo de constructor de movimiento

```
#include <iostream>
#include <cstring>
#include <vector>

using namespace std;

class Movible {
public:
    Movible(unsigned size) :
         buffer_(new char[size]),
         size_(size)
    {
    }

    Movible(const Movible& m)
    {
        cout << "Constructor de copia" << endl;
        if (this == &m)
            return;
        size_   = m.size_;
        buffer_ = new char[size_];
        memcpy(buffer_, m.buffer_, size_);
    }

    Movible(Movible&& m)
    {
        cout << "Constructor de movimiento" << endl;
        size_     = m.size_;
        buffer_   = m.buffer_;
        m.buffer_ = nullptr;
    }

    Movible& operator=(Movible&& m)
    {
        if (this == &m)
            return *this;

        cout << "Asignacion de movimiento" << endl;
        size_     = m.size_;
        buffer_   = m.buffer_;
        m.buffer_ = nullptr;

        return *this;
    }


    ~Movible()
    {
        cout << "Destructor" << endl;
        if (buffer_ == nullptr)
            cout << "--> con nullptr (moviendo)" << endl;

        delete [] buffer_;
    }


```

C21

```
58
59 private:
60     char*    buffer_;
61     unsigned size_;
62 };
63
64
65 Movible getM()
66 {
67     cout << "getM()" << endl;
68     Movible nuevo_objecto(20000);
69     return nuevo_objecto;
70 }
71
72 int main(int argc, char *argv[])
73 {
74     vector<Movible> v;
75
76     Movible k(234303);
77     k = getM();
78
79     v.push_back(Movible(4000));
80
81     return 0;
82 }
```

La salida del programa anterior es la siguiente:

```
$ ./move
getM()
Asignacion de movimiento
Destructor
--> con nullptr (moviendo)
Constructor de movimiento
Destructor
--> con nullptr (moviendo)
Destructor
Destructor
```

Cuando se usa la asignación, el objeto temporal que se genera para devolver por copia un objecto, es capturado por la asignación con movimiento y no se realiza ninguna copia extra.

La biblioteca estándar está preparada para el uso de constructores de movimiento, y como se ve en el ejemplo anterior, lo que en c++03 supondría una copia por cada `push_back()` en c++11 supone una llamada transparente al constructor de movimiento, evitando así todas las copias que se realizarían de otra forma.

Nótese como en los movimientos se ejecuta el destructor del objeto,

21.5.3. Cambios en la biblioteca de C++

Una de las adiciones más importantes a la STL es la inclusión de la mayoría del TR1. Así, plantillas como `auto_ptr` (ahora `unique_ptr`), `shared_ptr` y `weak_ptr` forman parte del estándar.

Generación de número aleatorios

C++11 introduce una nueva forma de generar números pseudo-aleatorios. La novedad que se introduce es que el generador se divide en dos partes, el motor y la distribución que se usa.

Los posible motores a utilizar son: `std::linear_congruential` (generador linear congruencial), `std::subtract_with_carry` (resta con acarreo) y `std::mersenne_twister`, que se representan con plantillas. Existen definiciones de tipo, para poder usarlas sin configurar cada parámetro de las mismas: `minstd_rand0` y `minstd_rand` (lineales), `mt19937` y `mt19937_64` (mersenne twister), y `ranlux24_base`, `ranlux48_base` y `ranlux24` (resta con acarreo).

Las distribuciones: `uniform_int_distribution`, `bernoulli_distribution`, `geometric_distribution`, `poisson_distribution`, `binomial_distribution`, `uniform_real_distribution`, `exponential_distribution`, `normal_distribution` y `gamma_distribution`.

En el siguiente ejemplo se muestra un posible uso, sacando la semilla del reloj del sistema en este caso.

```
1  #include <iostream>
2  #include <functional>
3  #include <random>
4  #include <sys/time.h>
5
6  using namespace std;
7
8  int main(int argc, char *argv[])
9  {
10   struct timeval now;
11   gettimeofday(&now, 0);
12
13   minstd_rand motor;
14   motor.seed(now.tv_usec);
15
16   uniform_int_distribution<int> dist(1,6);
17   uniform_int_distribution<int> dist_2(1,50);
18
19   int loto = dist(motor);              // Uso directo
20
21   auto generador  = bind(dist, motor); // Bind
22   int  valor_dado = generador();       // Uso "bindeado"
23
24   cout << loto << " : " << valor_dado << endl;
25
26   return 0;
27 }
```


Tablas Hash

Se introducen 4 tipos de tablas hash: `std::unordered_set`, `std::unordered_multiset`, `std::unordered_map` y `std::unordered_multimap`. La que se corresponde con el concepto tradicional en la que está representada por `std::unordered_map`. Ya que su uso es similar a `std::map` y al resto de contenedores asociativos antiguos, simplemente se muestra un ejemplo a continuación. Hay que incluir también la cabecera correspondiente (línea 2).

```
1  #include <iostream>
2  #include <unordered_map>
3
4  int main(int argc, char *argv[])
5  {
6    std::unordered_map<int, float> miHash;
7    miHash[13] = 1.1;
8    std::cout << miHash[13] << std::endl;
9    return 0;
```

```
10 }
```

Expresiones regulares

Una de las características nuevas más interesantes que se han añadido al estándar son las expresiones regulares. Para ello se utiliza un objeto `std::regex` para construir la expresión.

```
1
2   std::regex eRE1("\\b((c|C)ur)([^ ]*)");
```

Para almacenar las coincidencias será necesario utilizar un objeto `std::smatch` (si es una cadena de C, será `std::cmatch`).

```
1   std::smatch match;
```

Es posible buscar todas las coincidencias dentro de una cadena como se muestra a continuación.

```
1   const std::string entrada ("Curso de experto en videojuegos.
        Mucho curro.");
2
3   // Si hay coincidencias ...
4   auto aux = entrada;
5   while (std::regex_search( aux, match, eRE1)) {
6     cout << match.str() << endl;
7     aux = match.suffix().str();
8   }
```

Y saber si una cadena cumple la expresión regular así:

```
1   if (regex_match(entrada, eRE2))
2     cout << "[" << entrada << "] cumple la regex" << endl;
```

Las expresiones regulares, por defecto, se escribirán con la sintaxis ECMAScript.

Tuplas

C++11 da soporte a la creación de tuplas que contengan diferentes tipos. Para ello se utiliza la plantilla `std::tuple`. Como se ve en el ejemplo siguiente, para obtener los valores se utiliza la función *templatizada* `std::get()`.

```
1  #include <iostream>
2  #include <tuple>
3
4  using namespace std;
5
6  typedef tuple <string, int, float> tuplaPuntos;
7
8  int main(int argc, char *argv[])
9  {
10   tuplaPuntos p1("Bilbo", 20, 35.0);
11
12   cout << "El jugador "     << get<0>(p1)
13        << " ha conseguido " << get<2>(p1)
```

```
14         << " puntos en "      << get<1>(p1)
15         << " jugadas"         << endl;
16
17   return 0;
18 }
```

Otras características

Aparte de las características mencionadas en las secciones anteriores, C++11 incluye una biblioteca para el uso de *traits* para la metaprogramación, también envoltorios para poder utilizar referencias en plantillas, métodos uniformes para calcular el tipo devuelto en objetos funciones y soporte multitarea.

21.6. Plugins

En términos generales se denomina *plug-in* (o *add-on*) a cualquier componente que añade (o modifica) la funcionalidad de una aplicación principal integrándose con ella mediante un API proporcionando ex profeso. Los *plugins* son un mecanismo que se emplea habitualmente cuando se desea que programadores ajenos al desarrollo del proyecto matriz puedan integrarse con la aplicación. Ofrece algunas ventajas interesantes respecto a una aplicación monolítica:

- Reduce la complejidad de la aplicación principal.
- Permite experimentar con nuevas características, que si resultan de interés, más tarde se pueden integrar en la línea de desarrollo principal.
- Ahorra mucho tiempo a los desarrolladores de las extensiones puesto que no necesitan compilar el proyecto completo.
- Permite a empresas o colectivos concretos implementar funcionalidades a la medida de sus necesidades, que normalmente no serían admitidas en la aplicación principal.
- En entornos de código privativo, permite a los fabricantes distribuir parte del programa en formato binario, ya sea la aplicación central o alguno de los plugins. También ocurre cuando partes distintas tienen licencias diferentes.

Asumiendo que la aplicación principal esté escrita en un lenguaje compilado (como C++) se pueden distinguir tres mecanismos básicos que puede utilizar una aplicación para ofrecer soporte de plugins:

- Empotrar un interprete para un lenguaje dinámico, tal como Lua, Python o Scheme. Esta opción se estudia más adelante en el presente documento. Si la aplicación matriz está escrita en un lenguaje dinámico no se requiere normalmente ningún mecanismo especial más allá de localizar y cargar los plugins desde sus ficheros.
- Proporcionar un protocolo basado en mensajes para que la aplicación principal se pueda comunicar con los plugins. Este es el caso de OSGi y queda fuera el ámbito de este curso. Este tipo de arquitectura es muy versátil (los plugins pueden incluso estar escritos en distintos lenguajes) aunque resulta bastante ineficiente.

- Proporcionar un API binaria y cargar los plugins como bibliotecas dinámicas. Es la opción más eficiente ya que la única penalización ocurre en el momento de la carga. Esta sección se ocupa de describir este mecanismo.

21.6.1. Entendiendo las bibliotecas dinámicas

En el módulo 1 ya se mostró el proceso necesario para generar una biblioteca dinámica. Hasta ahora hemos utilizado las bibliotecas como contenedores de funcionalidad común que puede ser utilizada por los ejecutables sin más que indicarselo al montador (*linker*). Sin embargo las bibliotecas dinámicas en los sistemas operativos con formato de ejecutables ELF (Executable and Linkable Format) pueden servir para mucho más.

Una característica interesante de los ejecutables y bibliotecas ELF es que pueden tener símbolos no definidos, que son resueltos en tiempo de ejecución. Con las bibliotecas esta característica va más allá, hasta el punto de que no es necesario resolver todos los símbolos en tiempo de compilación. Veamos todo esto con ejemplos.

Hagamos un pequeño programa que utiliza una biblioteca.

Listado 21.106: El programa principal (`main.c`) simplemente usa una biblioteca

```
1 void mylib_func(const char* str, int val);
2
3 int main() {
4   mylib_func("test", 12345);
5   return 0;
6 }
```

E implementemos una biblioteca trivial.

Listado 21.107: La biblioteca (`mylib.c`) simplemente traza las llamadas

```
1 #include <stdio.h>
2
3 void mylib_func(const char* str, int val) {
4   printf("mylib_func %s %d\n", str, val);
5 }
```

Compilando el ejemplo como se indicó en el módulo 1 obtenemos el ejecutable y la biblioteca. Recordemos que toda la biblioteca debe ser compilada con la opción `-fPIC` para generar código independiente de posición.

```
$ gcc -shared -fPIC -o libmylib.so mylib.c
$ gcc -o main main.c -L. -lmylib
```

Para ejecutarlo hay que indicarle al sistema operativo que también tiene que buscar bibliotecas en el directorio actual. Para eso basta definir la variable de entorno `LD_LIBRARY_PATH`.

```
$ LD_LIBRARY_PATH=. ./main
mylib_func test 12345
```

Sin tocar para nada todo lo hecho hasta ahora, vamos a generar otra biblioteca dinámica con la misma función definida de otra forma.

Listado 21.108: Otra implementación de la biblioteca mínima (`mylib2.c`)

```
1 #include <stdio.h>
2
3 void mylib_func(const char* str, int val) {
4   printf("cambiada mylib_func %d %s\n", val, str);
5 }
```

Hemos cambiado ligeramente el mensaje, pero podríamos haber implementado algo completamente diferente. Ahora compilamos como una biblioteca dinámica, pero ni siquiera tenemos que seguir el convenio de nombres tradicional.

```
$ gcc -shared -fPIC -o ml2.so mylib2.c
```

Y volvemos a ejecutar el programa de una forma muy peculiar:

```
$ LD_PRELOAD=ml2.so LD_LIBRARY_PATH=. ./main
cambiada mylib_func 12345 test
```

¿Sorprendido? No hemos recompilado el programa, no hemos cambiado la biblioteca original, pero hemos alterado el funcionamiento. Esta técnica puede utilizarse para multitud de fines, desde la depuración (e.g. *ElectricFence*) hasta la alteración de los ejecutables para corregir *errores* cuando no se dispone del código fuente.

Lo que pasa tras el telón podemos analizarlo con herramientas estándar:

```
$ ldd main
    linux-vdso.so.1 =>  (0x00007fff701ff000)
    libmylib.so => not found
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f13043dd000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f130477c000)
```

Todos los ejecutables dinámicos están montados con la biblioteca `ld.so` o `ld--linux.so`. Se trata del montador dinámico. Obsérvese cómo se incluye en el ejecutable la ruta completa (última línea). Esta biblioteca se encarga de precargar las bibliotecas especificadas en `LD_PRELOAD`, buscar el resto en las rutas del sistema o de `LD_LIBRARY_PATH`, y de cargarlas. El proceso de carga en el ejecutable incluye resolver todos los símbolos que no estuvieran ya definidos.

Cuando desde la biblioteca dinámica es preciso invocar funciones (o simplemente utilizar símbolos) definidas en el ejecutable, éste debe ser compilado con la opción `-rdynamic`. Por ejemplo:

Listado 21.109: El programa principal define símbolos públicos

```
1  #include <stdio.h>
2
3  void mylib_func(const char* str, int val);
4  int main_i = 54321;
5
6  void main_func(int v) {
7    printf("main_func %d\n", v);
8  }
9
10 int main() {
11   mylib_func("test", 12345);
12   return 0;
13 }
```

Y la biblioteca llama a las funciones definidas en el ejecutable:

Listado 21.110: La biblioteca llama a una función definida en el programa

```
1 #include <stdio.h>
2
3 void main_func(int v);
4 extern int main_i;
5
6 void mylib_func(const char* str, int val) {
7   printf("mylib_func %d %s\n", val, str);
8   main_func(main_i);
9 }
```

Compilar este ejemplo solo cambia en la opción `-rdynamic`.

```
$ gcc -shared -fPIC -o libmylib3.so mylib3.c
$ gcc -rdynamic -o main2 main2.c -L. -lmylib3
```

Y al ejecutarlo como antes:

```
$ LD_LIBRARY_PATH=. ./main2
mylib_func 12345 test
main_func 54321
```

Si todas estas actividades son realizadas por una biblioteca (`ld.so`) no debería extrañar que esta funcionalidad esté también disponible mediante una API, para la carga explícita de bibliotecas desde nuestro programa.

21.6.2. Plugins con `libdl`

El modo más sencillo (aunque rudimentario) para implementar plugins es utilizar la biblioteca `libdl` cuyo nombre significa exactamente eso: *dynamic loading*. El API de esta biblioteca se encuentra en el fichero de cabecera `dlfcn.h` es bastante simple:

```
1 void* dlopen(const char* filename, int flag);
2 void* dlsym(void* handle, const char* symbol);
3 int dlclose(void* handle);
4 char* dlerror(void);
```

La utilidad de las funciones es sencilla:

- **`dlopen()`** abre una biblioteca dinámica (un fichero .so) y devuelve un manejador.
- **`dlsym()`** carga y devuelve la dirección de símbolo cuyo nombre se especifique como `symbol`.
- **`dlclose()`** le indica al sistema que ya no se va a utilizar la biblioteca y puede ser descargada de memoria.
- **`dlerror()`** devuelve una cadena de texto que describe el último error producido por cualquiera de las otras funciones de la biblioteca.

Vamos a seguir un ejemplo muy sencillo en las próximas secciones. El ejemplo está formado por un programa principal que tiene la lógica de registro de los plugins (`main.c`), una biblioteca estática (`liba`) y una (`libb`) que se cargará dinámicamente. Ambas bibliotecas tienen un fichero de cabecera (`a.h` y `b.h`) y dos ficheros de implementación cada una (`a.c`, `a2.c`, `b.c` y `b2.c`). La funcionalidad es absolutamente trivial y sirve únicamente para ilustrar la ejecución de las funciones correspondientes.

Listado 21.111: Biblioteca estática `liba`: `a.h`

```
1 #ifndef A_H
2 #define A_H
3
4 void a(int i);
5 int a2(int i);
6
7 #endif
```

Listado 21.112: Biblioteca estática `liba`: `a.c`

```
1 #include <stdio.h>
2 #include "a.h"
3
4 void a(int i) {
5   printf("a(%d) returns '%d'\n", i, a2(i));
6 }
```

Listado 21.113: Biblioteca estática `liba`: `a2.c`

```
1 #include "a.h"
2
3 int a2(int i) {
4   return i + 1;
5 }
```

Listado 21.114: Biblioteca dinámica `libb`: `b.h`

```
1 #ifndef B_H
2 #define B_H
3
4 void b(int i);
5 int b2(int i);
6
7 #endif
```

Listado 21.115: Biblioteca dinámica `libb`: `b.c`

```
1 #include <stdio.h>
2 #include "b.h"
3
4 void b(int i) {
5    printf("b(%d) returns '%d'\n", i, b2(i));
6 }
```

Listado 21.116: Biblioteca dinámica `libb`: `b2.c`

```
1 #include "b.h"
```

```
2
3 int b2(int i) {
4   return i * i;
5 }
```

Estas bibliotecas se construyen exactamente del mismo modo que ya se explicó en el capítulo «Herramientas de Desarrollo». Veamos como ejemplo el `Makefile` para `libb`:

Listado 21.117: Makefile para la compilación de `libb`

```
1 CC = gcc
2 CFLAGS = -Wall -ggdb -fPIC
3 LDFLAGS = -fPIC -shared
4
5 TARGET = libb.so.1.0.0
6
7 all: $(TARGET)
8
9 $(TARGET): b.o b2.o
10     $(CC) -Wl,-soname,libb.so.1.0.0 $(LDFLAGS) -o $@ $^
11
12 clean:
13     $(RM) *.o *~ *.a $(TARGET)
```

Carga explícita

En primer lugar veamos cómo cargar y ejecutar un símbolo (la función `b()`) de forma explícita, es decir, el programador utiliza `libdl` para buscar la biblioteca y cargar el símbolo concreto que desea:

Listado 21.118: Carga explícita de símbolos con `libdl`: `main.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <dlfcn.h>
4  #include "a.h"
5  #define LIBB_PATH "./dirb/libb.so.1.0.0"
6
7  void error() {
8    fprintf(stderr, dlerror());
9    exit(1);
10 }
11
12 int main() {
13   int i = 3;
14   void *plugin;
15   void (*function_b)(int);
16
17   if ((plugin = dlopen(LIBB_PATH, RTLD_LAZY)) == NULL)
18     error();
19
20   if ((function_b = dlsym(plugin, "b")) == NULL)
21     error();
22
23   printf("Results for '%d':\n", i);
24   a(i);
25   function_b(i);
26
27   dlclose(plugin);
28   return 0;
29 }
```

La diferencia más importante respecto al uso habitual de una biblioteca dinámica es que **no hay ninguna referencia** a `libb` en la construcción del programa `main.c` del listado anterior. Veamos el `Makefile` de la aplicación:

Listado 21.119: Carga explícita de símbolos con `libdl`: `Makefile`

```
CC = gcc
CFLAGS = -Wall -ggdb -Idira
LDFLAGS = -Ldira
LDLIBS = -la -ldl

all: libs main

main: main.o

libs:
    $(MAKE) -C dira
    $(MAKE) -C dirb

clean:
    $(RM) main *.o *~
    $(MAKE) -C dira clean
    $(MAKE) -C dirb clean
```

Carga implícita

Veamos ahora cómo construir un sencillo mecanismo que cargue automáticamente el símbolo en el momento de solicitar su uso.

Listado 21.120: Carga implícita de símbolos con `libdl`: `plugin.c`

```
typedef struct plugin {
  char* key;
  void (*function)(int);
  struct plugin* next;
} plugin_t;

static plugin_t* plugins;

void
plugin_register(char* key, void (*function)(int)) {
  plugin_t* p = (plugin_t*) malloc(sizeof(plugin_t));
  p->key = key;
  p->function = function;
  p->next = plugins;
  plugins = p;
  printf("** Plugin '%s' successfully registered.\n", key);
}

void
plugin_unregister(char* key) {
  plugin_t *prev = NULL, *p = plugins;

  while (p) {
    if (0 == strcmp(p->key, key))
      break;

    prev = p;
    p = p->next;
  }

  if (!p)
    return;

  if (prev)
```

```
35      prev->next = p->next;
36    else
37      plugins = p->next;
38
39    free(p);
40  }
41
42  static plugin_t*
43  plugin_find(char* key) {
44    plugin_t* p = plugins;
45    while (p) {
46      if (0 == strcmp(p->key, key))
47        break;
48
49      p = p->next;
50    }
51    return p;
52  }
53
54  void
55  call(char* key, int i) {
56    plugin_t* p;
57
58    p = plugin_find(key);
59    if (!p) {
60      char libname[PATH_MAX];
61      sprintf(libname, "./dir%s/lib%s.so", key, key);
62      printf("Trying load '%s'.\n", libname);
63      dlopen(libname, RTLD_LAZY);
64      p = plugin_find(key);
65    }
66
67    if (p)
68      p->function(i);
69    else
70      fprintf(stderr, "Error: Plugin '%s' not available.\n", key);
71  }
```

Los plugins (líneas 1–5) se almacenan en una lista enlazada (línea 7). Las funciones `plugin_register()` y `plugin_unregister()` se utilizan para añadir y eliminar plugins a la lista. La función `call()` (líneas 54–71) ejecuta la función especifica (contenida en el plugin) que se le pasa el parámetro, es decir, invoca una función a partir de su nombre[19]. Esa invocación se puede ver en la línea 10 del siguiente listado:

Listado 21.121: Carga implícita de símbolos con `libdl`: `main.c`

```
1  #include <stdio.h>
2  #include "plugin.h"
3  #include "a.h"
4
5  int main() {
6    int i = 3;
7
8    printf("Results for '%d':\n", i);
9    a(i);
10   call("b", i);
11   return 0;
12 }
```

Para que los plugins (en este caso `libb`) se registren automáticamente al ser cargados se requiere un pequeño truco: el «atributo» `constructor` (línea 9) que provoca que la función que lo tiene se ejecute en el momento de cargar el objeto:

[19] Este proceso se denomina enlace tardío (*late binding*) o *name binding*.

Listado 21.122: Carga implícita de símbolos con `libdl`: `b.c`

```
1  #include <stdio.h>
2  #include "../plugin.h"
3  #include "b.h"
4
5  void b(int i) {
6    printf("b(%d) returns '%d'\n", i, b2(i));
7  }
8
9  static void init() __attribute__((constructor));
10
11 static void init() {
12   plugin_register("b", &b);
13 }
```

Aunque este sistema es muy simple (intencionadamente) ilustra el concepto de la carga de símbolos bajo demanda desconocidos en tiempo de compilación. A partir de él es más fácil entender mecanismos más complejos puesto que se basan en la misma idea básica.

`(constructor)`

En C++ no es necesario indicar manualmente estos atributos, basta definir un constructor para una variable estática.

El atributo *constructor* indica que el símbolo al que va asociado debe almacenarse en una sección de la biblioteca reservada para el código de los constructores de variables estáticas. Estos constructores deben ejecutarse tan pronto como la biblioteca se carga en memoria. Análogamente, la sección *destructor* aglutina los destructores de las variables estáticas, que se invocan tan pronto como la biblioteca es cerrada.

21.6.3. Plugins con Glib `gmodule`

La biblioteca `glib` es un conjunto de utilidades, tipos abstractos de datos y otras herramientas de uso general y absolutamente portables. Es una biblioteca muy utilizada en los desarrollos del proyecto GNU. Un buen ejemplo de su uso es la biblioteca GTK y el entorno de escritorio GNOME (GNU Object Model Environment). Una de esas utilidades es GModule, un sistema para realizar carga dinámica de símbolos compatible con múltiples sistemas operativos, incluyendo Sun, GNU/Linux, Windows, etc.

GModule ofrece un API muy similar a `libdl` con funciones prácticamente equivalentes:

```
1 GModule* g_module_open(const gchar* file_name, GModuleFlags flags);
2 gboolean g_module_symbol(GModule* module, const gchar* symbol_name,
3                          gpointer* symbol);
4 gboolean g_module_close(GModule* module);
5 const gchar * g_module_error(void);
```

Carga explícita

El siguiente listado muestra cómo hacer la carga y uso de la función `b()`, equivalente al listado 21.118:

Listado 21.123: Carga explícita de símbolos con `GModule`: `main.c`

```
1  #include <stdio.h>
2  #include <glib.h>
3  #include <gmodule.h>
4  #include "a.h"
5
6  #define LIBB_PATH "./dirb/libb.so.1.0.0"
7
8  void error() {
9    g_error(g_module_error());
10 }
11
12 int main(){
13   int i = 3;
14   GModule* plugin;
15   void (*function_b)(int);
16
17   if ((plugin = g_module_open(LIBB_PATH, G_MODULE_BIND_LAZY)) ==
         NULL)
18     error();
19
20   if (!g_module_symbol(plugin, "b", (gpointer*)&function_b))
21     error();
22
23   printf("Results for '%d'.\n",i);
24   a(i);
25   function_b(i);
26
27   g_module_close(plugin);
28
29   return 0;
30 }
```

Carga implícita

Por último, este módulo implementa el sistema de registro y carga automática usando una tabla hash de glib para almacenar los plugins:

Listado 21.124: Carga explícita de símbolos con `GModule`: `plugin.c`

```
1  #include <stdio.h>
2  #include <gmodule.h>
3  #include <glib/ghash.h>
4  #include "a.h"
5
6  #ifndef PATH_MAX
7  #define PATH_MAX 1024
8  #endif
9
10 static GHashTable* plugins = NULL;
11
12 void
13 plugin_register(char* key, void (*f)(int)) {
14   if (plugins == NULL)
15     plugins = g_hash_table_new_full(g_str_hash, g_str_equal, g_free
           , g_free);
16   g_hash_table_insert(plugins, key, f);
17   g_message("Plugin '%s' succesfully registered.", key);
```

```
18 }
19
20 void
21 plugin_unregister(const char* key) {
22   if (plugins != NULL)
23     g_hash_table_remove(plugins, key);
24 }
25
26 void
27 call(const char* key, int i) {
28   void (*p)(int) = NULL;
29
30   if (plugins != NULL)
31     p = g_hash_table_lookup(plugins, key);
32
33   if (!p) {
34     char libname[PATH_MAX];
35
36     sprintf(libname, "./dir%s/lib%s.so", key, key);
37     g_message("Trying load '%s'.", libname);
38     if (g_module_open(libname, G_MODULE_BIND_LAZY) == NULL)
39       g_error("Plugin '%s' not available", libname);
40
41     if (plugins != NULL)
42       p = g_hash_table_lookup(plugins, key);
43   }
44
45   if (!p)
46     g_error("Plugin '%s' not available", key);
47
48   p(i);
49 }
```

21.6.4. Carga dinámica desde Python

El módulo `ctypes`, de la librería estándar de Python, permite mapear los tipos de datos de C a Python para conseguir una correspondencia binaria. Eso hace posible cargar funciones definidas en librerías dinámicas creadas con C/C++ y utilizarlas directamente desde Python.

El siguiente listado muestra cómo cargar y utilizar la misma función `b()` de la librería dinámica de las secciones anteriores:

Listado 21.125: Carga de símbolos desde Python con `ctypes`

```
1 LIBB_PATH = "./dirb/libb.so.1.0.0"
2
3 import ctypes
4
5 plugin = ctypes.cdll.LoadLibrary(LIBB_PATH)
6 plugin.b(3)
```

21.6.5. Plugins como objetos mediante el patrón *Factory Method*

Los plugins implican la adición y eliminación de código en tiempo de ejecución. Los problemas asociados tienen mucho que ver con los problemas que resuelven muchos de los patrones que ya conocemos. En esta sección veremos una pequeña selección.

Recordemos el patrón *factory method* ya descrito en el módulo 1. Se basa en la definición de una interfaz para crear instancias de objetos, permitiendo que las subclases redefinan este método. Este patrón se utiliza frecuentemente acoplado con la propia jerarquía de objetos, de forma parecida al patrón *prototype*, dando lugar a lo que se conoce como *constructor virtual*. Veamos un ejemplo similar al que poníamos para ilustrar el patrón prototipo, pero ahora empleando el patrón *factory method*.

Listado 21.126: Ejemplo de patrón *factory method*

```
class weapon {
public:
    typedef shared_ptr<weapon> weapon_ptr;
    virtual weapon_ptr make() = 0;
    virtual void shoot() = 0;
    virtual ~weapon() {}
};
```

Empleamos `shared_ptr` para simplificar la gestión de la memoria y definimos un destructor virtual por si acaso alguna de las subclases necesitan liberar memoria dinámica. Ahora cualquier instancia de `rifle` podría ser usada como factoría, pero para simplificar aún más su uso vamos a definir una factoría que actúe de fachada frente a todos los *factory method* concretos. De esta forma disponemos de una factoría extensible.

Listado 21.127: Ejemplo de factoría extensible de armamento

```
class weapon_factory {
public:
    typedef shared_ptr<weapon> weapon_ptr;

    weapon_ptr make(const string& key) {
        weapon* aux = factories_[key];
        if (aux)
      return aux->make();

        return 0;
    }

    void reg(const string& key, weapon* proto) {
        factories_[key] = proto;
    }

    void unreg(const string& key) {
        factories_.erase(key);
    }

protected:
    map<string,weapon*> factories_;
};
```

Para añadir o eliminar nuevas subclases de `weapon` tenemos que llamar a `reg()` o `unreg()` respectivamente. Esto es adecuado para la técnica RAII en la que la creación y destrucción de un objeto se utiliza para el uso y liberación de un recurso:

Listado 21.128: Ejemplo de RAII para registro de nuevas armas

```
template <class weapon_type>
class weapon_reg {
    weapon_factory& factory_;
    const char* key_;
public:
    weapon_reg(weapon_factory& factory, const char* key)
```

```
7           : factory_(factory), key_(key) {
8           factory_.reg(key_, new weapon_type());
9       }
10      ~weapon_reg() {
11          factory_.unreg(key_);
12      }
13  };
```

Tanto la factoría como los objetos de registro podrían ser también modelados con el patrón *singleton*, pero para simplificar el ejemplo nos limitaremos a instanciarlos sin más:

Listado 21.129: Instancias de la factoría extensible y una factoría concreta

```
1 using namespace std::placeholders;
```

Veamos cómo ha quedado el ejemplo. Tenemos subclases derivadas de `weapon` que saben cómo construir nuevos elementos. Tenemos una factoría extensible que se puede poblar con nuevas subclases de `weapon`. Y finalmente tenemos una clase auxiliar para facilitar la extensión de la factoría con cualquier subclase de `weapon`. Es una estructura ideal para los plugins. Un plugin simplemente tiene que proporcionar nuevas subclases de `weapon` e instanciar un `weapon_reg` por cada una de ellas. Para ello tan solo habría que cambiar el método `make()` de la factoría:

Listado 21.130: Ejemplo de factoría extensible con plugins

```
1  class dynamic_weapon_factory : public weapon_factory {
2  public:
3      weapon_ptr make(const string& key) {
4          weapon_ptr ret = weapon_factory::make(key);
5          if (ret)
6              return ret;
7          load_plugin(key);
8          return weapon_factory::make(key);
9      }
10
11 private:
12     void load_plugin(const string& key) {
13         string libname = "./fmethod-" + key + ".so";
14         dlopen(libname.c_str(), RTLD_LAZY);
15     }
16 };
```

El código de un plugin es completamente análogo al de las otras factorías concretas, como `rifle`.

Listado 21.131: Ejemplo de plugin para la factoría extensible

```
1  #include "fmethod.hh"
2
3  class bow: public weapon {
4  public:
5      weapon_ptr make() { return weapon_ptr(new bow); }
6      void shoot() { cout << "shoot arrow." << endl; }
7  };
8
9  extern dynamic_weapon_factory dfactory;
10 weapon_reg<bow> bow_reg(dfactory, "bow");
```

La variable `dfactory` es la instancia de la factoría dinámica extensible. Está declarada en el programa principal, así que para poder ser utilizada desde una biblioteca es preciso que el *linker* monte el programa principal con la opción `-rdynamic`. Por último pondremos un ejemplo de uso de la factoría:

Listado 21.132: Ejemplo de uso de la factoría

```
1 void
2 load_and_shoot(const string name) {
3    shared_ptr<weapon> w = dfactory.make(name);
4    if (w)
5        w->shoot();
6    else
7        cout << "Missing weapon " << name << endl;
8 }
```

La descarga de la biblioteca dinámica (por ejemplo, utilizando la función `dlclose`) provocaría que se llamara al destructor de la clase `bow_factory` y con ello que se des-registrara la factoría concreta.

Nótese que en este ejemplo empleamos la infraestructura de plugins para mantener extensible nuestra aplicación, pero no manejamos explícitamente los plugins. Así, por ejemplo, no hemos proporcionado ninguna función de descarga de plugins. Incidiremos en este aspecto en el siguiente ejemplo.

21.6.6. Plugins multi-plataforma

La biblioteca GModule que hemos visto en la sección es compatible con múltiples sistemas operativos. Sin embargo, no está todo resuelto automáticamente. Es preciso conocer algunos detalles de las plataformas más comunes para poder implantar con éxito una arquitectura de plugins. Para ello veremos una adaptación del ejemplo anterior para ejecutables PE (ReactOS, Microsoft Windows).

En el caso de los ejecutables PE no es posible compilar bibliotecas (DLL (Dynamic Link Library)) sin determinar las referencias a todos los símbolos. Por tanto no es posible referirnos a un símbolo definido en el programa principal (EXE). La solución más sencilla es extraer la parte común del ejecutable en una biblioteca dinámica que se monta tanto con el ejecutable como con las otras bibliotecas. El programa principal queda reducido a:

Listado 21.133: Programa principal para Windows

```
1  #include "fmethod-win.hh"
2
3  extern dynamic_weapon_factory dfactory;
4
5  int main(int argc, char* argv[]) {
6      while (argc > 1) {
7          shared_ptr<weapon> w = dfactory.make(argv[1]);
8          if (w)
9           w->shoot();
10         else
11              cout << "Missing weapon " << argv[1] << endl;
12
13         argc--; ++argv;
14     }
15 }
```

Y la parte común se extraería en:

Listado 21.134: Biblioteca común con la factoría para Windows

```
1  #include "fmethod-win.hh"
2  #include <windows.h>
3
4  void
5  dynamic_weapon_factory::load_plugin(const string& key)
6  {
7      string libname = "./fmethod-" + key + "-win.dll";
8      LoadLibrary(libname.c_str());
9  }
10
11 dynamic_weapon_factory dfactory;
12 weapon_reg<rifle> rifle_reg(dfactory, "rifle");
```

Nótese cómo se cargan las bibliotecas con `LoadLibrary()`. El plugin es muy similar a la versión ELF:

Listado 21.135: Plugin para Windows

```
1  #include "fmethod-win.hh"
2
3  class bow: public weapon {
4  public:
5      weapon_ptr make() { return weapon_ptr(new bow); }
6      void shoot() { cout << "shoot arrow." << endl; }
7  };
8
9  extern dynamic_weapon_factory dfactory;
10 weapon_reg<bow> bow_reg(dfactory, "bow");
```

Para compilar y probar todo no es necesario utilizar ReactOS o Microsoft Windows. Podemos usar el compilador cruzado GCC para MINGW32 y el emulador `wine`.

```
$ i586-mingw32msvc-g++ -std=c++0x -shared \
  -Wl,--enable-runtime-pseudo-reloc \
  -o fmethod-fac-win.dll fmethod-fac-win.cc
$ i586-mingw32msvc-g++ -std=c++0x \
  -Wl,--enable-runtime-pseudo-reloc \
  -Wl,--enable-auto-import -o fmethod-win.exe \
  fmethod-win.cc fmethod-fac-win.dll
$ i586-mingw32msvc-g++ -std=c++0x -shared
  -Wl,--enable-runtime-pseudo-reloc \
  -o fmethod-bow-win.dll fmethod-bow-win.cc \
  fmethod-fac-win.dll
$ wine fmethod-win.exe rifle bow 2>/dev/null
```

La opción del montador `-enable-runtime-pseudo-reloc` permite utilizar la semántica tradicional de visibilidad de símbolos de Unix. Todos los símbolos externos son automáticamente exportados. La opción `-enable-auto-import` permite que todos los símbolos usados en el ejecutable que no están definidos en el propio ejecutable sean automáticamente importados.

Se propone como ejercicio la generalización de este código para que el mismo programa compile correctamente con ejecutables ELF o con ejecutables PE.

Capítulo 22

Técnicas específicas

Francisco Moya Fernández
Félix J. Villanueva Molina
Sergio Pérez Camacho
David Villa Alises

22.1. Serialización de objetos

La serialización de objetos tiene que ver en parte con la persistencia del estado de un videojuego. Serializar un objeto consiste en convertirlo en algo almacenable y recuperable. De esto modo, el estado completo del objeto podrá ser escrito en disco o ser enviado a través de la red, y su estado podrá ser recuperado en otro instante de tiempo o en otra máquina.

Puede parecer una operación sencilla, después de todo, bastaría con almacenar el pedazo de memoria que representa al objeto y volver a ponerlo en el mismo sitio después. Lamentablemente esto no es posible, puesto que la configuración de la memoria varía de ejecución en ejecución y de máquina en máquina. Además, cada objeto tiene sus particularidades. Por ejemplo, si lo que se desea serializar es una `std::string` seguramente sea suficiente con almacenar los caracteres que la componen.

Uno de los problemas a la hora de serializar objetos es que estos pueden contener referencias o punteros a otros objetos, y este estado ha de conservarse de forma fidedigna. El problema de los punteros, es que la direcciones de memoria que almacenan serán diferentes en cada ejecución.

Antes de hablar de la serialización propiamente dicha, se presentarán los *streams* de C++.

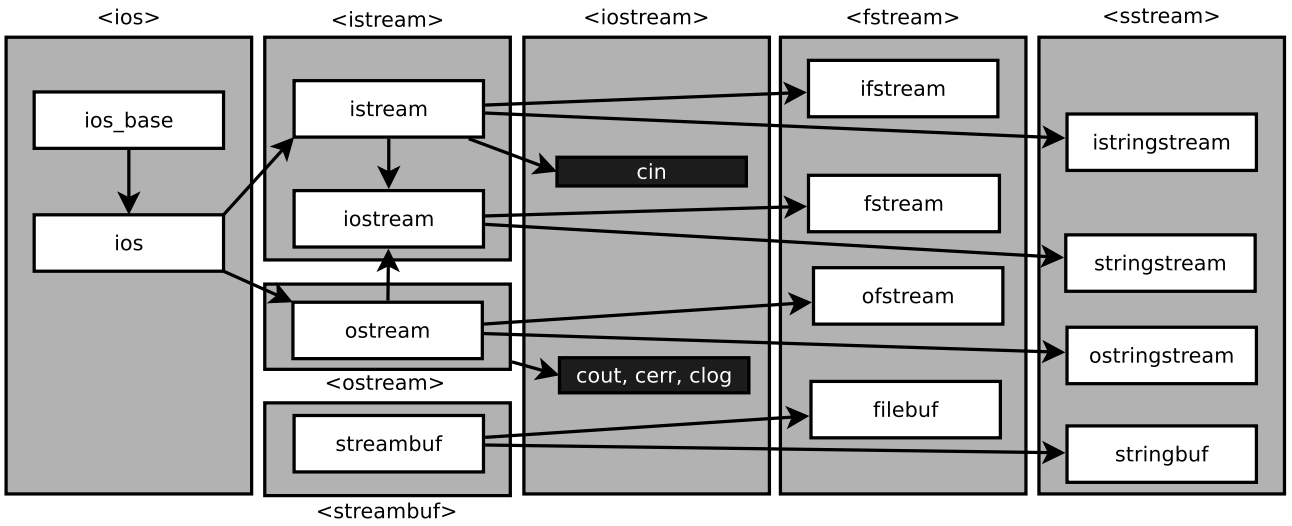


Figura 22.1: Jerarquía de *streams*

22.1.1. *Streams*

Un *stream* es como una tubería por donde fluyen datos. Existen *streams* de entrada o de salida, y de ambas, de modo que un programa puede leer (entrada) de una abstrayéndose por completo de qué es lo que está llenando la misma. Esto hace que los *streams* sean una forma de desacoplar las entradas de la forma de acceder a las mismas, al igual que las salidas. No importa si quien rellena un *stream* es la entrada del teclado o un archivo, la forma de utilizarla es la misma para ambos casos. De este modo, controlar la entrada supondría conectar un *stream* a un fichero (o al teclado) y su salida al programa. Justo al revés (donde el teclado sería ahora la pantalla) se controlaría la salida.

Normalmente los *streams* tienen un *buffer* asociado puesto que escribir o leer en bloques suele ser mucho más eficiente en los dispositivos de entrada y salida. El *stream* se encargará (usando un `streambuf`[1]) de proporcionar o recoger el número de bytes que se requiera leer o escribir en el mismo

En la figura 22.1 se muestra la jerarquía de *streams* en la biblioteca estándar de C++. La clase `ios_base` representa la propiedades generales de un *stream*, como por ejemplo si este es de entrada o de salida o si es de texto o binaria. La clase `ios`, que hereda de la anterior, contiene un `streambuf`. Las clases `ostream` y `istream`, derivan de `ios` y proporcionan métodos de salida y de entrada respectivamente.

`istream`

La clase `istream` implementa métodos que se utilizan para leer del *buffer* interno de manera transparente. Existen dos formas de recoger la entrada: formateada y sin formatear. La primera usa el operador $>>$ y la segunda utiliza los siguientes miembros de la clase:

[1] `streambuf` es una clase que provee la memoria para dicho *buffer* incluyendo además funciones para el manejo del mismo (rellenado, *flushing*, etc...)

gcount	Devuelve el número de caracteres que retornó la última lectura no formateada
get	Obtiene datos sin formatear del *stream*
getline	Obtiene una línea completa del *stream*
ignore	Saca caracteres del *stream* y los descarta
peek	Lee el siguiente carácter sin extraerlo del *stream*
read	Lee en bloque el número de caracteres que se le pidan
readsome	Lee todo lo disponible en el *buffer*
putback	Introduce de vuelta un carácter en el *buffer*
unget	Decrementa el puntero *get*. Se leerá de nuevo el mismo carácter.

Utilizando `tellg` se obtiene la posición (`streampos`) del puntero en el *stream*, y es posible modificar la misma utilizando `seekg` con la posición que de desee como entrada. La función `seekg` también se puede utilizar con un *offset* como primer parámetro y con una posición base como segundo. Así, `ios_base::beg`, `ios_base::cur` y `ios_base::end` representan al principio del *stream*, a la posición actual y al final del mismo respectivamente. Es posible (y de hecho necesario con `end`) utilizar números negativos para posicionarse en un *stream*.

ostream

Un `ostream` representa una tubería en la que se puede escribir. Al igual que un `istream`, se soportan los datos formateados, en este caso la inserción, usando el operador $<<$.

Las operaciones para datos no formateados son las siguientes:

put	Escribe un carácter en el *stream*
write	Escribe un conjunto de caracteres desde un *buffer*

ifstream y ofstream

Estos *streams* que se utilizan para leer y escribir de archivos.

En el ejemplo siguiente se muestra cómo leer de un archivo utilizando los visto sobre *streams*.

Listado 22.1: Ejemplo de lectura de un archivo

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4
5  using namespace std;
6
7  int main(int argc, char *argv[])
8  {
9    ifstream infile("prueba.txt", ios_base::binary);
10
11   if (!infile.is_open()) {
12     cout << "Error abriendo fichero" << endl;
13     return -1;
14   }
15
16   string linea;
17   getline(infile, linea);
18   cout << linea << endl;
19
20   char buffer[300];
```

```
21   infile.getline(buffer, 300);
22   cout << buffer << endl;
23
24   infile.read(buffer,3);
25   buffer[3] = '\0';
26   cout << "[" << buffer << "]" << endl;
27
28   streampos p = infile.tellg();
29   infile.seekg(2, ios_base::cur);
30   infile.seekg(-4, ios_base::end);
31   infile.seekg(p);
32
33   int i;
34   while ((i = infile.get()) != -1)
35     cout << "\'" << (char) i << "\'=int(" << i << ")" << endl;
36
37   return 0;
38 }
```

En la línea [9] de crea el *stream* del fichero, y se intenta abrir para lectura como un fichero binario. En [11-14] se comprueba que el archivo se abrió y se termina el programa si no es así. En [16-18] se usa una función global de `string` para rellenar una de estas con una línea desde el fichero. Se hace lo mismo con un *buffer* limitado a 300 caracteres en la líneas [20-22]. Después se leen 3 caracteres sueltos (sin tener en cuenta el final de linea) ([24-26]). En [28-31] se juega con la posición del puntero de lectura, y en el resto, se lee carácter a carácter hasta el final del archivo.

Los modos de apertura son los siguientes:

in	Permitir sacar datos del *stream*
out	Permitir introducir datos en el *stream*
ate	Al abrir el *stream*, situar el puntero al final del archivo.
app	Poner el puntero al final en cada operación de salida
trunc	Trunca el archivo al abrirlo
binary	El *stream* será binario y no de texto

En el listado 22.2 se muestra cómo copiar un archivo.

Operadores de inserción y extracción

Es posible definir (sobrecargar) los operadores de inserción o de extracción para cualquier clase que nos interese, y así poder utilizarla para rellenar un *stream* o para modificarla extrayendo datos de un *stream*. Estos operadores se usan para una entrada/salida formateada. Vea el listado 22.3.

En la líneas [15-19] se define el operador de inserción, y en las líneas [21-32] el de extracción. Es necesario definir estos operadores como amigos de la clase (líneas [6-7]) ya que necesitan acceso a los atributos privados.

La forma de utilizarlos es la siguiente:

Listado 22.4: Operadores de inserción y extracción

```
1 int main(int argc, char *argv[])
2 {
3   Vector3D v(1.0, 2.3, 4.5);
4   cout << v << endl;
5   cin  >> v ;
6   cout << v << endl;
7   return 0;
8 }
```

Listado 22.2: Ejemplo de copia desde un archivo a otro

```
1  #include <fstream>
2  using namespace std;
3
4  int main()
5  {
6    fstream in ("prueba.txt", ios_base::in  | ios_base::binary);
7    fstream out("copiaP.txt", ios_base::out | ios_base::binary
8                                            | ios_base::trunc );
9    if (!in.is_open() || !out.is_open())
10     return -1;
11
12   in.seekg(0, ios_base::end);
13   size_t size = in.tellg();
14   in.seekg(0, ios_base::beg);
15
16   char* buffer = new char[size];
17
18   in.read  (buffer,  size);
19   out.write(buffer, size);
20
21   delete [] buffer;
22   return 0;
23 }
```

El programa anterior imprime el valor original del vector, y espera a la entrada de un vector con el mismo formato. Al pulsar **RETURN** el vector original se rellenará con los nuevos datos tomados de la entrada estándar. De hecho, el programa funciona también con una tubería del tipo `echo "(1.0, 2.912, 3.123)"| ./ejecutable`.

stringstream

La clase `stringstream` proporciona un interfaz para manipular cadenas como si fueran *streams* de entrada/salida.

Su uso puede sustituir de algún modo al de `sprintf`, ya que es posible utilizar un objeto de este tipo para transformar números en cadenas y para realizar un formateo básico (ver listado 22.5).

En las líneas 6-12 se define una función *templatizada* que se usa en 33 para transformar un número en una cadena, usando `streamstream` e invocando luego su método `str()`, que devuelve la cadena asociada.

En 14-23 se define otra que se puede utilizar para extraer un número de una cadena. Se ve un ejemplo de uso en la línea 34.

22.1.2. Serialización y Dependencias entre objetos

A la hora de serializar un objeto, o un conjunto de objetos, se pueden dar diferentes escenarios. No es lo mismo tener que escribir el contenido de un objeto que no tiene ninguna dependencia con otros, que tener que escribir el contenido de un conjunto de objetos que dependen unos de otros.

Listado 22.3: Operadores de inserción y extracción de `Vector3D`

```
#include <iostream>

using namespace std;

class Vector3D {
  friend ostream& operator<<(ostream& o, const Vector3D& v);
  friend istream& operator>>(istream& i,Vector3D& v);
public:
  Vector3D(float x, float y, float z) :
    x_(x), y_(y), z_(z) {}
private:
  float x_, y_, z_;
};

ostream& operator<<(ostream& o, const Vector3D& v)
{
  o << "(" << v.x_ << ", " << v.y_ << ", " << v.z_ << ")" ;
  return o;
}

istream& operator>>(istream& i,Vector3D& v)
{
  char par, coma;
  // formato: (X, Y, Z)
  i >> par;
  i >> v.x_;
  i >> coma;
  i >> v.y_;
  i >> coma;
  i >> v.z_;
  return i;
}
```

Sin dependencias

El escenario más simple es la serialización de un objeto sin dependencias con el resto, es decir, un objeto que no apunta a ningún otro y que está autocontenido.

La serialización será entonces trivial, y bastará con escribir cada una de los valores que contenga, y recuperarlo en el mismo orden.

Sea la siguiente una interfaz para objetos que puedan serializarse.

Listado 22.6: Interfaz simple para objetos serializables

```
class ISerializable {
public:
  virtual void read (std::istream& in)  = 0;
  virtual void write(std::ostream& out) = 0;
};
```

De este modo, todos los objetos que deriven de esta clase tendrán que implementar la forma de escribir y leer de un *stream*. Es útil delegar los detalles de serialización al objeto.

Supóngase ahora una clase muy sencilla y sin dependencias, con un `double`, un `int` y un `string` para serializar.

Listado 22.5: Usando un `stringstream`

```
1  #include <iostream>
2  #include <sstream>
3
4  using namespace std;
5
6  template<typename T>
7  string toString(T in)
8  {
9    stringstream ss;
10   ss << in;
11   return ss.str();
12 }
13
14 template<typename T>
15 T toNumber(const string& s)
16 {
17   stringstream ss(s);
18   T t;
19   ss << s;
20   if (!(ss >> t))
21     throw;
22   return t;
23 }
24
25
26 int main(int argc, char *argv[])
27 {
28   stringstream s;
29
30   s << 98 << endl << "texto" << endl;
31   cout << (s.str() += "op\n") ;
32
33   string str = toString(9.001);
34   long   a = toNumber<long>("245345354525");
35   cout << a << endl;
36   return 0;
37 }
```

Listado 22.7: Objeto serializable sin dependencias

```
1  class ObjetoSimple : public ISerializable
2  {
3  public:
4    ObjetoSimple(double a, int b, std::string cad);
5
6    ObjetoSimple();
7    virtual ~ObjetoSimple();
8
9    virtual void read (std::istream& in);
10   virtual void write(std::ostream& out);
11
12 private:
13
14   double      a_;
15   int         b_;
16   std::string cad_;
17 };
```

La implementación de `read()` y `write()` sería como sigue:

Listado 22.8: Detalle de implementación de un serializable simple

```
1  void
2  ObjetoSimple::read(std::istream& in)
3  {
4    in.read((char*) &a_, sizeof(double));
5    in.read((char*) &b_, sizeof(int));
6
7    size_t len;
8    in.read((char*) &len, sizeof(size_t));
9    char* auxCad = new char[len+1];
10
11   in.read(auxCad, len);
12   auxCad[len] = '\0';
13   cad_ = auxCad;
14
15   delete [] auxCad;
16
17   std::cout << "a_: "   << a_   << std::endl;
18   std::cout << "b_: "   << b_   << std::endl;
19   std::cout << "cad_: " << cad_ << std::endl;
20 }
21
22 void
23 ObjetoSimple::write(std::ostream& out)
24 {
25   out.write((char*) &a_, sizeof(double));
26   out.write((char*) &b_, sizeof(int));
27
28   size_t len = cad_.length();
29   out.write((char*) &len, sizeof(size_t));
30   out.write((char*) cad_.c_str(), len);
31 }
```

En la lectura y escritura se realiza un *cast* a `char*` puesto que así lo requieren las funciones `read` y `write` de un *stream*. Lo que se está pidiendo a dichas funciones es: "desde/en esta posición de memoria, tratada como un `char*`, lee/escribe el siguiente número de caracteres".

El número de caracteres (bytes/octetos en x86+) viene determinado por el segundo parámetro, y en este ejemplo se calcula con `sizeof`, esto es, con el tamaño del tipo que se está guardando o leyendo.

Un caso especial es la serialización de un `string`, puesto que como se aprecia, no se está guardando todo el objeto, sino los caracteres que contiene. Hay que tener en cuenta que será necesario guardar la longitud de la misma (línea 30) para poder reservar la cantidad de memoria correcta al leerla de nuevo (8-9).

A continuación se muestra un ejemplo de uso de dichos objetos utilizando archivos para su serialización y carga.

Listado 22.9: Uso de un objeto serializable simple

```
1  int main(int argc, char *argv[])
2  {
3    {
4      ofstream fout("data.bin", ios_base::binary | ios_base::trunc);
5      if (!fout.is_open())
6        return -1;
7
8      ObjetoSimple o(3.1371, 1337, "CEDV");
9      o.write(fout);
10     ObjetoSimple p(9.235, 31337, "UCLM");
11     p.write(fout);
12
13   }
14
```

```
15   ifstream fin("data.bin", ios_base::binary);
16   ObjetoSimple q;
17   q.read(fin);
18   ObjetoSimple r;
19   r.read(fin);
20
21   return 0;
22 }
```

Se está utilizando un archivo para escribir el valor de un par de objetos, y tras cerrarse, se vuelve a abrir para leer los datos almacenados y rellenar un par nuevo.

Con dependencias

Habrá dependencia entre objetos, cuando la existencia de uno esté ligada a la de otro. Normalmente esto viene determinado porque uno de los miembros de una clase es un puntero a la misma o a otra clase.

Cuando existen objetos con dependencias hay dos aproximaciones posibles para su serialización. La primera consiste en diseñar la arquitectura para que no se utilicen punteros. En vez de esto se utilizarán UUID (Universally Unique Identifier)s (IDs únicas universales). Un objeto, en vez de almacenar un puntero al resto de objetos, almacenará su UUID y hará uso de factorías para recuperar el objeto en tiempo de carga o de ejecución. Las ventajas son claras, y las desventajas son el tiempo necesario para mantener las referencias actualizadas, y que la arquitectura dependerá de esta decisión de diseño completamente.

Otra forma de serializar clases con punteros es escribir sin preocupación y reparar el estado no-válido de ese objeto teniendo en cuenta las propiedades de los mismos. Un puntero referencia una dirección de memoria única, es decir, dos objetos diferentes no podrán compartir la misma dirección de memoria. Visto de otro modo, dos punteros iguales apuntan al mismo objeto. Teniendo esto en cuenta, el propio puntero podría valer como un UUID interno para la serialización.

De este modo, la serialización y deserialización lectura de objetos con punteros podría ser del siguiente modo:

- Almacenar todos los objetos, teniendo en cuenta que lo primero que se almacenará será la dirección de memoria que ocupa el objeto actual. Los punteros del mismo se almacenarán como el resto de datos.
- Al leer los objetos, poner en una tabla el puntero antiguo leído, asociado a la nueva dirección de memoria.
- Hacer una pasada corrigiendo el valor de los punteros, buscando la correspondencia en la tabla.

Para ello necesitamos una interfaz nueva, que soporte la nueva función `fixPtrs()` y otras dos para leer y recuperar la posición de memoria del propio objeto.

Listado 22.10: Nueva interfaz de objeto serializable

```
1 class ISerializable {
2 public:
3   virtual void read (std::istream& in)  = 0;
4   virtual void write(std::ostream& out) = 0;
5
6   virtual void fixPtrs () = 0;
7
```

```
 8 protected:
 9   virtual void readMemDir (std::istream& in)   = 0;
10   virtual void writeMemDir(std::ostream& out)  = 0;
11 };
```

Esta vez se implementará dicha interfaz con la clase `Serializable`:

Listado 22.11: Implementación de la interfaz ISerializable

```
 1 class Serializable : public ISerializable {
 2 public:
 3   Serializable();
 4   ~Serializable();
 5
 6   virtual void read (std::istream& in)  = 0;
 7   virtual void write(std::ostream& out) = 0;
 8
 9   virtual void fixPtrs () = 0;
10
11 protected:
12   virtual void readMemDir (std::istream& in);
13   virtual void writeMemDir(std::ostream& out);
14
15   Serializable* sPtr;
16 };
```

En la línea [15] se añade un puntero que almacenará la dirección de memoria de la propia clase.

La implementación de las funciones de lectura y escritura se muestra a continuación.

Listado 22.12: Implementación de la interfaz ISerializable (II)

```
 1 void Serializable::readMemDir(std::istream& in)
 2 {
 3   in.read((char*) &sPtr, sizeof(Serializable*) );
 4   LookUpTable::getMap()[sPtr] =  this;
 5 }
 6
 7 void Serializable::writeMemDir (std::ostream& out)
 8 {
 9   sPtr = this;
10   out.write((char*) &sPtr, sizeof(Serializable*) );
11 }
```

Cuando se lee la antigua dirección de memoria en `readMemDir`, esta se almacena en una tabla junto con la nueva dirección (línea [4]). La implementación de la tabla se podría dar a través de una especie de *Singleton*, que envolvería un `map` y lo mostraría como una variable global.

Listado 22.13: Tabla de búsqueda de punteros

```
 1 class Serializable; // Forward Dec.
 2
 3 class LookUpTable
 4 {
 5   friend class std::auto_ptr<LookUpTable*>;
 6 public:
 7   static std::map<Serializable*, Serializable*>& getMap();
 8
 9   typedef std::map<Serializable*, Serializable*>::iterator itMapS;
10
```

```
11 private:
12   LookUpTable(){}
13
14   std::map<Serializable*, Serializable*> sMap_;
15
16 };
```

Listado 22.14: Tabla de búsqueda de punteros (II)

```
1 std::map<Serializable*, Serializable*>&
2 LookUpTable::getMap()
3 {
4   static std::auto_ptr<LookUpTable> instance_(new LookUpTable);
5   return instance_->sMap_;
6 }
```

El nuevo tipo de objeto compuesto tendrá que derivar de la clase `Serializable` y no de `ISerializable` como antes.

Listado 22.15: Declaración de `ObjetoCompuesto`

```
1  class ObjetoCompuesto : public Serializable
2  {
3  public:
4    ObjetoCompuesto(double a, int b, std::string cad,
5                    ObjetoCompuesto* other);
6
7    ObjetoCompuesto();
8    virtual ~ObjetoCompuesto();
9
10   virtual void read (std::istream& in);
11   virtual void write(std::ostream& out);
12
13   virtual void fixPtrs();
14
15   void printCad();
16   void printOther();
17
18 private:
19
20   double      a_;
21   int         b_;
22   std::string cad_;
23   ObjetoCompuesto* obj_;
24 };
```

Uno de los constructores ahora acepta un puntero a un objeto del mismo tipo. En 23 se declara un puntero a un objeto del mismo tipo, y tendrá que ser serializado, recuperado y arreglado. Con motivo de probar si la lectura ha sido correcta, se han añadido un par de funciones, `printCad`, que imprime la cadena serializada del propio objeto y `printOther`, que imprime la cadena del objeto apuntado a través del primer método.

De esto modo, la implementación de la clase anterior sería la siguiente. Primero para las funciones de impresión, que son las más sencillas:

Listado 22.16: Definición de `ObjetoCompuesto`

```
1 void ObjetoCompuesto::printCad()
2 {
3     std::cout << cad_ << std::endl;
4 }
5
```

```
6 void ObjetoCompuesto::printOther()
7 {
8   if (obj_) obj_->printCad();
9 }
```

Y a continuación las de serialización y deserialización, con el añadido de que justo antes de leer el resto del objeto, se lee la dirección de memoria que se almacenó (línea 4), que será la encargada de rellenar la tabla de punteros como se ha visto anteriormente. En la línea 19 se lee el puntero, como se haría de forma normal. En este momento, el puntero contendría la dirección antigua fruto de la serialización. Para la escritura pasa exactamente lo mismo, simplemente se guardan los punteros que corresponden a las direcciones de memoria en el momento de la escritura.

Listado 22.17: Definición de `ObjetoCompuesto` (II)

```
1 void
2 ObjetoCompuesto::read(std::istream& in)
3 {
4   readMemDir(in);
5
6   in.read((char*) &a_, sizeof(double));
7   in.read((char*) &b_, sizeof(int));
8
9   size_t len;
10  in.read((char*) &len, sizeof(size_t));
11  char* auxCad = new char[len+1];
12
13  in.read(auxCad, len);
14  auxCad[len] = '\0';
15  cad_ = auxCad;
16
17  delete [] auxCad;
18
19  in.read((char*) &obj_, sizeof(ObjetoCompuesto*) );
20
21  std::cout << "a_: "   << a_   << std::endl;
22  std::cout << "b_: "   << b_   << std::endl;
23  std::cout << "cad_: " << cad_ << std::endl;
24  std::cout << "obj_: " << obj_ << std::endl;
25  std::cout << "this: " << this << std::endl;
26  std::cout << "---------------" << std::endl;
27 }
28
29 void
30 ObjetoCompuesto::write(std::ostream& out)
31 {
32  writeMemDir(out);
33
34  out.write((char*) &a_, sizeof(double));
35  out.write((char*) &b_, sizeof(int));
36
37  size_t len = cad_.length();
38  out.write((char*) &len, sizeof(size_t));
39  out.write((char*) cad_.c_str(), len);
40
41  out.write((char*) &obj_, sizeof(ObjetoCompuesto*) );
42  std::cout << "* obj_: " << obj_ << std::endl;
43 }
```

La función que se encarga de arreglar los punteros es la siguiente:

Listado 22.18: Definición de `ObjetoCompuesto` (III)

```
1 void ObjetoCompuesto::fixPtrs() {
2   if (obj_ == NULL)
```

```
3     return;
4
5   LookUpTable::itMapS it;
6   it = LookUpTable::getMap().find(obj_);
7   if (it == LookUpTable::getMap().end()) {
8     std::cout << "Puntero no encontrado" << std::endl;
9     throw;
10  }
11  obj_ = (ObjetoCompuesto*) it->second;
12  std::cout << "obj_ FIXED: " << obj_ << std::endl;
13 }
```

Si el puntero almacenado es nulo, no cambiará nada. Si el puntero no es nulo, se sustituirá por el que esté almacenado en la tabla, que será precisamente la nueva posición del objeto apuntado en memoria. Hay que tener en cuenta que para que esta función no falle, primero tendrá que estar cargado en memoria en objeto al que se debería estar apuntando.

Así, una forma de utilizar todas estas clases sería esta:

Listado 22.19: Serialización con dependencias

```
1  int main() {
2    cout << "Serializando" << endl; cout << "------------" << endl;
3    {
4      ofstream fout("data.bin", ios_base::binary | ios_base::trunc);
5      if (!fout.is_open())
6        return -1;
7
8      ObjetoCompuesto o(3.1371, 1337,  "CEDV", NULL);
9      o.write(fout);
10     ObjetoCompuesto p(9.235,  31337, "UCLM", &o);
11     p.write(fout);
12     ObjetoCompuesto q(9.235,  6233,  "ESI",  &p);
13     q.write(fout);
14
15     ObjetoCompuesto* k = new ObjetoCompuesto(300.2,  1000,  "BRUE",
           &p);
16     k->write(fout);
17     delete k;
18
19     ObjetoCompuesto r(10.2,   3243,  "2012", k);
20     r.write(fout);
21   }
22   cout << "\nRecuperando" << endl;
23   cout << "-----------" << endl;
24
25   ifstream fin("data.bin", ios_base::binary);
26
27   std::vector<Serializable*> objetosLeidos;
28
29   for (int i = 0; i < 5; ++i) {
30     ObjetoCompuesto* o = new ObjetoCompuesto();
31     o->read(fin);
32     objetosLeidos.push_back(o);
33   }
34
35   cout << "\nFix punteros" << endl;
36   cout << "------------" << endl;
37
38   for_each(objetosLeidos.begin(), objetosLeidos.end(),
39            mem_fun(&Serializable::fixPtrs));
40
41   cout << "\nProbando" << endl;
42   cout << "--------"   << endl;
43
44   std::vector<Serializable*>::iterator it;
45   for (it = objetosLeidos.begin();
```

```
46        it != objetosLeidos.end();
47        ++it)
48     static_cast<ObjetoCompuesto*>((*it))->printOther();
49
50   return 0;
51 }
```

En las líneas 5-23 se crea el archivo que se usará como un *stream* y algunos objetos que se van serializando. Algunos de ellos se crean en el *stack* y otro en el *heap*. El archivo se cerrará puesto que la variable `fout` sale de contexto al terminar el bloque.

En la línea 27 se abre el mismo archivo para proceder a su lectura. En 29-35 se leen los datos del archivo y se van metiendo en un `vector`. En 40-41 se procede a ejecutar la función `fixPtrs` de cada uno de los objetos almacenados dentro del vector. Justo después se ejecutan las funciones que imprimen las cadenas de los objetos apuntados, para comprobar que se han restaurado correctamente las dependencias.

La salida al ejecutar el programa anterior se muestra a continuación:

```
Serializando
------------
* obj_: 0
* obj_: 0x7fff3f6dad80
* obj_: 0x7fff3f6dadb0
* obj_: 0x7fff3f6dadb0
* obj_: 0x11b3320

Recuperando
-----------
a_: 3.1371
b_: 1337
cad_: CEDV
obj_: 0
this: 0x11b3260
--------------
a_: 9.235
b_: 31337
cad_: UCLM
obj_: 0x7fff3f6dad80
this: 0x11b3370
--------------
a_: 9.235
b_: 6233
cad_: ESI
obj_: 0x7fff3f6dadb0
this: 0x11b3440
--------------
a_: 300.2
b_: 1000
cad_: BRUE
obj_: 0x7fff3f6dadb0
this: 0x11b3520
--------------
a_: 10.2
b_: 3243
cad_: 2012
obj_: 0x11b3320
this: 0x11b35d0
--------------
```

```
Fix punteros
------------
obj_ FIXED: 0x11b3260
obj_ FIXED: 0x11b3370
obj_ FIXED: 0x11b3370
obj_ FIXED: 0x11b3520

Probando
--------
CEDV
UCLM
UCLM
BRUE
```

Cabe destacar que la dirección de memoria obtenida de los objetos en el *stack* se diferencia notablemente de la obtenida del *heap*. Como se puede ver, la serialización y la posterior lectura es correcta cuando se arreglan los punteros con la técnica presentada.

22.1.3. Serialización con Boost

Boost provee al programador de C++ con muchas utilidades, entre ellas la capacidad para serializar objetos de forma muy sencilla y metódica, convirtiendo una tarea tediosa en un mero trámite.

Objetos sin dependencias

Para serializar la clase simple expuesta en la sección anterior, primero habría del siguiente modo:

Listado 22.20: Serializando un objeto simple con Boost

```
 1 #include <fstream>
 2 #include <boost/archive/text_oarchive.hpp>
 3 #include <boost/archive/text_iarchive.hpp>
 4
 5 class ObjetoSimple {
 6 friend class boost::serialization::access;
 7 public:
 8   ObjetoSimple(double a, int b, std::string cad);
 9   ObjetoSimple();
10   virtual ~ObjetoSimple();
11
12   void print();
13
14   template<class Archive>
15   void serialize(Archive & ar, const unsigned int version) {
16     ar & a_;
17     ar & b_;
18     ar & cad_;
19   }
20
21 private:
22   double      a_;
23   int         b_;
24   std::string cad_;
25 };
```

En la línea [8] se permite el acceso a esta clase desde la función `access` de Boost, que se usará para la invocación de `serialize` [18-22]. El símbolo & utilizado dentro de dicha función *templatizada* representa a << o >> según sea el tipo de `Archive`, que será el envoltorio de `fstreams` de Boost usado para la serialización. Es precisamente en esa función donde se lleva a cabo la serialización, puesto que para cada variable de la clase, se procede a su lectura o escritura.

A continuación se muestra cómo utilizar esta clase en un programa:

Listado 22.21: Uso de un objeto simple serializable con Boost

```
1   {
2     ofstream fout ("dataSimple", ios_base::trunc);
3     ObjetoSimple oSimple(1.0, 2, "BOOST");
4     boost::archive::text_oarchive outA(fout);
5     outA << oSimple;
6   }
7
8   {
9     ObjetoSimple otherSimple;
10    ifstream fin("dataSimple", ios_base::binary );
11    boost::archive::text_iarchive inA(fin);
12    inA >> otherSimple;
13    otherSimple.print();
14  }
```

En el primer bloque se crea un archivo de salida, y se crean y escriben dos objetos. En el segundo se leen y se imprimen. Como se muestra en la líneas [5] y [12], se usan los operadores de inserción y extracción de las clases de Boost utilizadas.

Objetos con dependencias

Sea la siguiente clase una similar a la compuesta que se planteó en la sección anterior, añadiendo además un objeto de tipo `ObjetoSimple` como miembro.

Listado 22.22: Declarando un objeto compuesto serializable con Boost

```
1  class ObjetoCompuesto
2  {
3  friend class boost::serialization::access;
4  public:
5    ObjetoCompuesto(double a, int b, std::string cad,
6                    ObjetoCompuesto* other);
7
8    ObjetoCompuesto();
9    virtual ~ObjetoCompuesto();
10
11   void print();
12   void printOther();
13
14   template<class Archive>
15   void serialize(Archive & ar, const unsigned int version) {
16     ar & a_;
17     ar & b_;
18     ar & cad_;
19     ar & simple_;
20     ar & obj_;
21   }
22
23 private:
24   double      a_;
25   int         b_;
26   std::string cad_;
27
```

```
28   ObjetoSimple     simple_;
29   ObjetoCompuesto* obj_;
30 };
```

Como se puede apreciar, la serialización se lleva a cabo de la misma manera si se utiliza Boost. De hecho la forma de utilizarlos es similar, excepto a la hora de crear los objetos:

Listado 22.23: Uso de un objeto compuesto serializable

```
1   {
2     ofstream fout ("dataCompuesto", ios_base::trunc );
3     ObjetoCompuesto oComp (4.534,  90, "BOOST COMPO", NULL);
4     ObjetoCompuesto oComp2(43.234, 280, "OTRO BOOST COMPO", &oComp)
          ;
5     boost::archive::text_oarchive outA(fout);
6     outA << oComp;
7     outA << oComp2;
8   }
9
10  {
11    ObjetoCompuesto otherComp;
12    ObjetoCompuesto otherComp2;
13
14    ifstream fin("dataCompuesto", ios_base::binary );
15    boost::archive::text_iarchive inA(fin);
16
17    inA >> otherComp;
18    inA >> otherComp2;
19
20    otherComp.print();
21    cout << "\n\n\n";
22    otherComp2.print();
23  }
```

De hecho, dos de los pocos casos donde esta forma difiere se muestran en el siguiente apartado.

Objetos derivados y con contenedores

En el código siguiente se muestra una clase `Base` y una clase `ObjetoDerivadoCont` que hereda de ella. Además, incluye un contenedor `vector` que se serializará con la misma.

Listado 22.24: Declarando un objeto base serializable con Boost

```
1  class Base {
2  friend class boost::serialization::access;
3  public:
4    Base(const std::string& bName) :
5      baseName_(bName) {}
6
7    virtual void print() {
8      std::cout << "Base::print(): " << baseName_;
9    };
10
11   virtual ~Base() {}
12
13   template<class Archive>
14   void serialize(Archive & ar, const unsigned int version) {
15     ar & baseName_;
16   }
17
```

```
18 protected:
19   std::string baseName_;
20 };
```

Listado 22.25: Declarando un objeto derivado y con contenedores serializable con Boost

```
1  class ObjetoDerivadoCont : public Base
2  {
3  friend class boost::serialization::access;
4  public:
5    ObjetoDerivadoCont(std::string s) :
6      Base(s) { }
7
8    ObjetoDerivadoCont() : Base("default") {}
9
10   virtual ~ObjetoDerivadoCont(){}
11
12   virtual void print();
13
14   void push_int(int i) {
15     v_.push_back(i);
16   };
17
18   template<class Archive>
19   void serialize(Archive & ar, const unsigned int version) {
20     ar & boost::serialization::base_object<Base>(*this);
21     ar & v_;
22   }
23
24 private:
25   std::vector<int> v_;
26 };
```

La única cosa que hay que tener en cuenta a la hora de serializar este tipo de clases es que hay que ser explícito a la hora de serializar la parte relativa a la clase base. Esto se lleva a cabo como se muestra en la línea [20] del código anterior.

Para que se puedan serializar contenedores, simplemente habrá que incluir la cabecera de Boost correspondiente:

Listado 22.26: Cabecera de Boost para serializar `vector`

```
1 #include <boost/serialization/vector.hpp>
```

Si se quisiera serializar una `list`, se usaría `list.hpp`.

A continuación, se muestra un ejemplo de uso donde se ve cómo se rellenan los `vectors`, para luego serilizar dos los objeto y proceder a recuperarlos en el segundo bloque.

Listado 22.27: Uso de un objeto derivado y con contenedores

```
1    {
2      ofstream fout ("dataDerivadoCont", ios_base::trunc);
3      boost::archive::text_oarchive outA(fout);
4
5      ObjetoDerivadoCont oDeriv ("DERIVADO1");
6      oDeriv.push_int(38);  oDeriv.push_int(485);
7      oDeriv.push_int(973); oDeriv.push_int(545);
8
9      ObjetoDerivadoCont oDeriv2("DERIVADO2");
10     oDeriv2.push_int(41356);  oDeriv2.push_int(765);
11
```

```
12      outA << oDeriv;
13      outA << oDeriv2;
14    }
15
16    {
17      ifstream fin("dataDerivadoCont", ios_base::binary );
18      boost::archive::text_iarchive inA(fin);
19
20      ObjetoDerivadoCont oD;
21      ObjetoDerivadoCont oD2;
22
23      inA >> oD;
24      inA >> oD2;
25
26      oD.print();
27      cout << "\n\n\n";
28      oD2.print();
29      cout << "\n\n\n";
30    }
```

Con todos los ejemplos anteriores se puede afrontar casi cualquier tipo de serialización. Queda claro que el uso de Boost acelera el proceso, pero aun existen plataformas donde Boost no está portada (aquellas con compiladores que no soportan todas las características de C++, por ejemplo) y donde la STL aun lucha por parecerse al estándar. Es en éstas donde habrá que realizar una serialización más artesana y usar algún tipo de técnica parecida a la vista en las primeras secciones.

22.2. C++ y scripting

A pesar de que el uso de un lenguaje de propósito general como C++ nos permite abordar cualquier tipo de problema, existen lenguajes mas o menos apropiados para tareas específicas. En el diseño de un lenguaje se tiene en mente aspectos como la eficiencia, portabilidad, simpleza, etc. y difícilmente se pueden alcanzar la excelencia en todas las facetas.

No obstante, sería deseable que pudiéramos realizar cada tarea en aquel lenguaje mas apropiado para la tarea a realizar. Por ejemplo, mientras que C/C++ se caracterizan, entre otras cosas, por su eficiencia, lenguajes como Python nos proporcionan un entorno de programación simple y muy productivo de cara a prototipado rápido así como una gran portabilidad.

Existen muchos proyectos que utilizan varios lenguajes de programación, utilizando el mas apropiado para cada tarea. En esta sección vamos a ver un ejemplo de esta interacción entre diversos lenguajes de programación. En concreto vamos a coger C++, como ya hemos comentado, un lenguaje orientado a objetos muy eficiente en su ejecución y Python, un lenguaje interpretado (como java, php, Lua etc.) muy apropiado por su simpleza y portabilidad que nos permite desarrollar prototipos de forma rápida y sencilla.

22.2.1. Consideraciones de diseño

En el caso de juegos, el planteamiento inicial es qué partes implementar en C++ y qué partes dejar al lenguaje de *scripting*.

En el caso del desarrollo de juegos cuyo lenguaje principal sea de *scripting* (por ejemplo, Python), una aproximación genérica sería, desarrollar el juego por completo, y después, mediante técnicas de *profiling* se identifican aquellas partes críticas para mejorar las prestaciones, que son las que se implementan en C/C++. Obviamente aquellas partes que, a priori, ya sabemos que sus prestaciones son críticas podemos anticiparnos y escribirlas directamente en C/C++.

En el caso de que la aplicación se implemente en C/C++, utilizamos un lenguaje de *scripting* para el uso de alguna librería concreta o para poder modificar/adaptar/extender/corregir el comportamiento sin tener que recompilar. En general, cuando hablamos de C++ y *scripting* hablamos de utilizar las características de un lenguaje de prototipado rápido desde C++, lo cual incluye, a grandes rasgos:

Lua vs Python

Mientras que Lua está pensado para extender aplicaciones y como lenguaje de configuración, Python es mas completo y puede ser utilizado para funciones mas complejas.

- Crear y borrar objetos en el lenguaje de *scripting* e interaccionar con ellos invocando métodos .
- pasar datos y obtener resultados en invocaciones a funciones
- Gestionar posibles errores que pudieran suceder en el proceso de interacción, incluyendo excepciones.

Otro ejemplo de las posibilidades de los lenguajes de scripting son utilizar lenguajes específicos ampliamente usados en otros entornos como la inteligencia artificial, para implementar las partes relacionadas del juego. Ejemplos de este tipo de lenguajes serían LISP y Prolog ampliamente usados en inteligencia artificial, y por lo tanto, muy apropiados para modelar este tipo de problemas.

En la actualidad, las decisiones de diseño en cuanto a qué lenguaje de *scripting* usar viene determinado por las características de dichos lenguajes. Sin tener en cuenta lenguajes muy orientados a problemas concretos como los mencionados LISP y Prolog, y considerando sólo aquellos lenguajes de *scripting* de propósito general, las opciones actuales pasan por Lua y Python principalmente.

Atendiendo a sus características, Python:

- Tiene una gran librería y, generalmente, bien documentada.
- Facilita la gestión de cadenas y tiene operadores binarios.
- A partir de la versión 2.4, Python tiene los denominados *ctypes* que permiten acceder a tipos de librerías compartidas sin tener que hacer un *wrapper* C.
- Tiene buenas prestaciones en computación numérica (lo cual es muy deseable en simuladores de eventos físicos)

En contraste Lua es un lenguaje mas simple, originalmente pensado para labores de configuración y que ha sido orientado específicamente a la extensión de aplicaciones, algunas de sus características son:

- En general, usa menos memoria y el intérprete es mas rápido que el de Python.
- Tiene una sintaxis simple y fácil de aprender si bien es cierto que no tiene la documentación, ejemplos y tutoriales que Python.

Es cierto que tanto Lua como Python pueden ser utilizados para extender aplicaciones desarrolladas en C/C++, la decisión de qué lenguaje usar depende de qué características queremos implementar en el lenguaje de *scripting*. Al ser Python un lenguaje mas genérico, y por tanto mas versátil, que Lua será el que estudiaremos mas en profundidad.

22.2.2. Invocando Python desde C++ de forma nativa

Uso de lenguajes compilados

Aquellas partes de cálculo intensivo deben ir implementadas en los lenguajes eficientes (compilados)

En nomenclatura Python, hablamos de extender Python cuando usamos funciones y objetos escritos en un lenguaje (por ejemplo C++) desde programas en Python. Por el contrario, se habla de Python embebido cuando es Python el que se invoca desde una aplicación desarrollada en otro lenguaje. Desde la nomenclatura C/C++ se habla de *scripting* cuando accedemos a un lenguaje de script desde C++.

El interprete Python ya incluye extensiones para empotrar Python en C/C++. Es requisito imprescindible tener instalado en la máquina a ejecutar los ejemplos de esta sección, el intérprete de Python (usaremos la versión 2.7) aunque dichas extensiones están desde la versión 2.2.

En el primer ejemplo, vamos a ver la versión Python del intérprete y que nos sirve para ver cómo ejecutar una cadena en dicho intérprete desde un programa en C++.

Listado 22.28: Imprimiendo la versión del intérprete Python desde C++

```
1 #include <python2.7/Python.h>
2
3 int main(int argc, char *argv[])
4 {
5   Py_Initialize();
6   PyRun_SimpleString("import sys; print '%d.%d' % sys.version_info
        [:2]\n");
7   Py_Finalize();
8   return 0;
9 }
```

La función `Py_Initialize()` inicializa el intérprete creando la lista de módulos cargados (sys.modules), crea los módulos básicos (__main__, __builtin__ y sys) y crea la lista para la búsqueda de módulos sys.path. En definitiva lo prepara para recibir órdenes.`PyRun_SimpleString()` ejecuta un comando en el intérprete, podemos ver que en este caso, importamos el módulo `sys` y a continuación imprimimos la versión del intérprete que estamos ejecutando. Por último, finalizamos la instancia del intérprete liberando la memoria utilizada y destruyendo los objetos creados en la sesión.

Todas estas funciones se definen en el archivo `Python.h` que proporciona la instalación de Python y que proporciona un API para acceder al entorno de ejecución de este lenguaje. El propio intérprete de Python utiliza esta librería.

Estas extensiones permiten invocar todo tipo de sentencias e interaccionar con el intérprete de Python, eso si, de forma no muy orientada a objetos. Utilizando el tipo *PyObject* (concretamente punteros a este tipo) podemos obtener referencias a cualquier módulo e invocar funciones en ellas. En la tabla 22.1 podemos ver, de forma muy resumida, algunas funciones que nos pueden ser muy útiles. Por supuesto no están todas pero nos pueden dar una referencia para los pasos principales que necesitaríamos de cara a la interacción C++ y Python.

La gestión de errores (del módulo sys) en la actualidad está delegada en la función `exc_info()()` que devuelve una terna que representan el tipo de excepción que se ha producido, su valor y la traza (lo que hasta la versión 1.5 representaban las variables sys.exc_type, sys.exc_value y sys.exc_traceback).

Con el ejemplo visto en esta subsección no existe un intercambio entre nuestro programa C++ y el entorno Python. Por supuesto, el soporte nativo de Python nos permite realizar cualquier forma de interacción que necesitemos. No obstante, podemos beneficiarnos de librerías que nos hacen esta interacción mas natural y orientada a objetos. Vamos a estudiar la interacción entre ambos entornos mediante la librería boost.

Función	Cometido
Py_Initialize()	Inicializa el intérprete
PyString_FromString("cadena")	Retorna un puntero a PyObject con una cadena (E.j. nombre del módulo a cargar).
PyImport_Import(PyObject* name)	Carga un módulo, retorna un puntero a PyObject.
PyModule_GetDict(PyObject* modulo)	Obtiene el diccionario con atributos y métodos del módulo. Retorna un puntero a PyObject.
PyDict_GetItemString(PyObject *Diccionario, "función")	Obtiene una referencia a una función. Retorna un puntero a PyObject
PyObject_CallObject(PyObject *función, argumentos)	Llama a la función con los argumentos proporcionados.
PyCallable_Check(PyObject *funcion)	Comprueba que es un objeto invocable.
PyRun_File	Interpreta un archivo
PyTuple_New(items)	Crea una tupla
PyTuple_SetItem(tupla, posición, item)	Almacena un Item en una tupla
PyErr_Print()	Imprime error.
PyList_Check(PyObject*)	Comprueba si PyObject es una lista

Cuadro 22.1: Funciones útiles de invocación de Python desde C++

22.2.3. Librería boost

La librería boost [1] nos ayuda en la interacción de C++ y Python. Es necesario resaltar que está mas evolucionada en el uso de C++ desde Python que al revés. Esto es debido a que generalmente, es un caso de uso mas frecuente el usar C++ desde Python por dos motivos principalmente:

- Aumentar la eficiencia del programa implementando partes críticas en C++.
- Usar alguna librería C++ para la cual no existen *bindings* en Python.

No obstante, como ya hemos indicado anteriormente, el uso de Python desde C++ también cuenta con ventajas y para introducir la librería boost, vamos a continuar con nuestro ejemplo de obtener la versión del interprete desde nuestro programa en C++.

Usando Python desde nuestro programa en C++

Nuestra primera modificación va a ser imprimir la versión del intérprete desde C++, por lo que debemos realizar un intercambio de datos desde el intérprete de Python al código en C++.

Listado 22.29: Obteniendo información del intérprete Python desde C++

```
1 #include <boost/python.hpp>
2 #include <boost/python/import.hpp>
3 #include <iostream>
4
5 using namespace boost::python;
6 using namespace std;
7
8 int main(int argc, char *argv[])
9 {
```

```
10
11    Py_Initialize();
12    PyRun_SimpleString("import sys; major, minor = sys.version_info
          [:2]");
13    object mainobj = import("__main__");
14    object dictionary = mainobj.attr("__dict__");
15    object major = dictionary["major"];
16    int major_version = extract<int>(major);
17    object minor = dictionary["minor"];
18    int minor_version = extract<int>(minor);
19    cout<<major_version<<"."<<minor_version<<endl;
20    Py_Finalize();
21    return 0;
22  }
```

Debemos observar varios puntos en este nuevo listado:

- Seguimos usando *Py_Initialize* y *Py_Finalize*. Estas funciones se utilizan siempre y son obligatorias, en principio, no tienen equivalente en boost.
- Se usa Run_SimpleString para seguir con el ejemplo anterior, luego veremos como substituir esta sentencia por usos de la librería boost.
- Para acceder al interprete de Python, necesitamos acceder al módulo principal y a su diccionario (donde se definen todos los atributos y funciones de dicho módulo). Este paso se realiza en las lineas 13 y 14.
- Una vez obtenemos el diccionario, podemos acceder a sus variables obteniéndolas como referencias a `object()`, linea 15.

La plantilla `extract()` nos permite extraer de una instancia de *object*, en principio, cualquier tipo de C++. En nuestro ejemplo extraemos un entero correspondiente a las versiones del intérprete de Python (versión mayor y menor). De forma genérica, si no existe una conversión disponible para el tipo que le pasamos a `extract()`, una excepción Python (*TypeError*) es lanzada.

Como vemos en este ejemplo, la flexibilidad de Python puede simplificarnos la interacción con la parte de C++. La sentencia (linea 12) `sys.version_info` nos devuelve un tupla en Python, no obstante, hemos guardado esa tupla como dos enteros (*major* y *minor*) al cual accedemos de forma individual (líneas 16 y 19 mediante *extract*). Como ya hemos comentado, esta plantilla es clave de cara a obtener referencias a los tipos básicos desde C++ y puede ser empleado para aquellos tipos básicos definidos como pueden ser std::string, double, float, int, etc. Para estructuras mas complejas (por ejemplo, tuplas), esta extracción de elementos se puede realizar mediante el anidamiento de llamadas a la plantilla *extract*.

Modificando brevemente el ejemplo anterior podemos mostrar el caso más básico de una tupla. tal y como podemos ver en este listado:

Listado 22.30: Extracción de tipos compleja

```
1   PyRun_SimpleString("import sys; result = sys.version_info[:2]");
2   object mainobj = import("__main__");
3   object dictionary = mainobj.attr("__dict__");
4   object result = dictionary["result"];
5   tuple tup = extract<tuple>(result);
6   if (!extract<int>(tup[0]).check() || !extract<int>(tup[1]).check
        ())
7     return 0;
8   int major =extract<int>(tup[0]);
9   int minor =extract<int>(tup[1]);
10  cout<<major<<"."<<minor<<endl;
```

```
11    Py_Finalize();
12    return 0;
13  }
```

ahora vemos como guardamos en *result* la tupla que, posteriormente, es guardada en la variable *tup* mediante el `extract()` correspondiente (línea 5).

A partir de este punto podemos obtener los elementos de la tupla (obviamente conociendo de antemano los campos de dicha tupla y su disposición en la misma) como podemos ver en las líneas 8 y 9. Obviamente, es recomendable realizar la comprobación de que la conversión de un entorno a otro se ha realizado correctamente mediante el uso de la función `check()` (linea 6).

Para el siguiente ejemplo vamos a dar un paso mas allá en nuestra forma de pasar datos de un entorno a otro. Particularizando en la programación de videojuegos, vamos a suponer que tenemos una clase hero la cual, va a representar un héroe. Cada instancia coge su nombre del héroe que representa y a continuación se le asigna un arma.

Listado 22.31: Clase hero

```
1  class hero{
2    string _name;
3    string _weapon;
4    int amunnition;
5  public:
6    hero(){}
7    hero(string name){
8      _name=name;
9    }
10
11   void configure()
12   {
13     cout<<"Getting configuration:"<<_name<<": "<<_weapon<<endl;
14   }
15   void weapon(string weapon){
16     _weapon=weapon;
17   }
18 };
```

Además se tiene un método `configure()`, que nos permite obtener la configuración del héroe en concreto, en este caso, simplemente la imprime. Bien asumimos como decisión de diseño, que, salvo el nombre, el arma asignada a cada héroe será variable y podremos ir obteniendo diversas armas conforme avancemos en el juego. Esta última parte la decidimos implementar en Python. Por lo tanto, habrá un método en Python, al cual le pasaremos un objeto de la clase hero y ese método lo configurará de forma apropiada (en nuestro caso sólo con el tipo de arma). En el siguiente listado podemos ver esta función. En este ejemplo simplemente le pasa el arma (Kalasnikov) invocando el método correspondiente.

Listado 22.32: Configurar una instancia de la clase hero desde Python

```
1  def ConfHero(hero):
2      hero.weapon("Kalasnikov")
3      hero.configure()
```

Para conseguir este ejemplo, necesitamos exponer la clase hero al intérprete de Python.

En boost, se usa la macro BOOST_PYTHON_MODULE que básicamente crea un módulo (*ConfActors*), que podremos usar en Python, definiendo las clases y métodos que le proporcionemos (en nuestro caso el constructor que acepta una cadena y los métodos `configure()` y `weapon()`)

Listado 22.33: Exponer clases C++ a entornos Python

```
1 // Exposing class heroe to python
2 BOOST_PYTHON_MODULE( ConfActors )
3 {
4   class_<hero>("hero")
5     .def(init<std::string>() )
6     .def("configure", &hero::configure)
7     .def("weapon", &hero::weapon)
8   ;
9 }
```

Con esta infraestructura vamos a invocar la función en Python `ConfHero()` para que le asigne el arma y, a continuación vamos a comprobar que esa asignación se realiza de forma satisfactoria.

Listado 22.34: Pasando objetos C++ como argumentos de funciones en Python

```
1  int main(int argc, char *argv[])
2  {
3    Py_Initialize();
4    initConfActors(); //initialize the module
5    object mainobj = import("__main__");
6    object dictionary(mainobj.attr("__dict__"));
7    object result = exec_file("configureActors.py", dictionary,
         dictionary);
8    object ConfHero_function = dictionary["ConfHero"];
9    if(!ConfHero_function.is_none())
10     {
11       boost::shared_ptr<hero> Carpanta(new hero("Carpanta"));
12       ConfHero_function(ptr(Carpanta.get()));
13       hero *obj = ptr(Carpanta.get());
14       obj->configure();
15     }
16   Py_Finalize();
17   return 0;
18 }
```

En el listado anterior, en la línea 7 cargamos el contenido del archivo Python en el diccionario, con esta sentencia ponemos en el diccionario toda la información relativa a atributos y a funciones definidas en dicho archivo. A continuación ya podemos obtener un objeto que representa a la función Python que vamos a invocar (línea 8).

Si este objeto es válido (línea 9), obtenemos un puntero compartido al objeto que vamos a compartir entre el intérprete Python y el espacio C++. En este caso, creamos un objeto de la clase *hero* (línea 11).

Ya estamos listo para invocar la función proporcionándole la instancia que acabamos de crear. Para ello, utilizamos la instancia del puntero compartido y obtenemos con `get()` la instancia en C++, con el cual podemos llamar a la función (línea 13) y por supuesto comprobar que, efectivamente, nuestro héroe se ha configurado correctamente (línea 14).

Invocando C++ desde el intérprete Python

Veamos ahora el caso contrario, es decir, vamos a tener una clase en C++ y vamos a acceder a ella como si de un módulo en Python se tratara. De hecho el trabajo duro ya lo hemos realizado, en el ejemplo anterior, ya usábamos un objeto definido en C++ desde el interprete en Python.

Aprovechemos ese trabajo, si tenemos en un archivo el código relativo a la clase *hero* (listado 22.31) y la exposición realizada de la misma (listado 22.33) lo que nos falta es construir un módulo dinámico que el intérprete Python pueda cargar. En este punto nos puede ayudar el sistema de construcción del propio interprete. Efectivamente podemos realizar un archivo setup.py tal y como aparece en el listado 22.35

Listado 22.35: Configuración para generar el paquete Python a partir de los fuentes C++

```
1
2 from distutils.core import setup, Extension
3
4 module1 = Extension('ConfActors', sources = ['hero.cc'] , libraries
      = ['boost_python-py27'])
5
6 setup (name = 'PackageName',
7         version = '1.0',
8         description = 'A C++ Package for python',
9         ext_modules = [module1])
```

De esta forma, podemos decirle a las herramientas de construcción y distribución de paquetes Python toda la información necesaria para que haga nuestro nuevo paquete a partir de nuestros fuentes en C++. En él, se le indica los fuentes.

Para compilar y generar la librería que, con posterioridad, nos permitirá importarla desde el intérprete de comandos, debemos invocar el archivo setup.py con el intérprete indicándole que construya el paquete:

```
python setup.py build
```

Esto nos generará la librería específica para la máquina donde estamos y lo alojará en el directorio build que creará en el mismo directorio donde esté el setup.py (build/lib.linux-i686-2.7/ en nuestro caso) y con el nombre del módulo (ConfActors.so) que le hemos indicado. A partir de este punto, previa importación del módulo ConfActors, podemos acceder a todas sus clases y métodos directamente desde el interprete de python como si fuera un módulo mas escrito de forma nativa en este lenguaje.

22.2.4. Herramienta SWIG

No se puede terminar esta sección sin una mención explícita a la herramienta SWIG [3], una herramienta de desarrollo que permite conectar programas escritos en C/C++ con una amplia variedad de lenguajes de programación de scripting incluidos Python, PHP, Lua, C#, Java, R, etc.

Para C++ nos automatiza la construcción de *wrappers* para nuestro código mediante una definición de las partes a utilizar en el lenguaje destino.

A modo de ejemplo básico, vamos a usar una nueva clase en C++ desde el interprete Python, en este caso una clase player al cual le vamos a proporcionar parámetros de configuración.

Listado 22.36: Definición de la clase Player

```
1
2  #include <string>
3  #include <iostream>
4
5  class Player
6  {
7    std::string _name;
8    std::string _position;
9   public:
10   Player(std::string name);
11   void position(std::string pos);
12   void printConf();
13 };
```

Y su implementación:

Listado 22.37: Implementación de la clase en C++

```
1
2  using namespace std;
3  Player::Player(string name){
4    _name=name;
5  }
6
7  void Player::position(string pos){
8    _position=pos;
9  }
10
11 void Player::printConf(){
12   cout<<_name<<" "<<_position<<endl;
13 }
```

Sin modificación de estos archivos construimos un archivo de configuración para swig:

Listado 22.38: Archivo de configuración de SWIG

```
1  #define SWIG_FILE_WITH_INIT
2  #include "player.h"
3  %}
4
5  %include "std_string.i"
6  %include "player.h"
```

Con este archivo de configuración generamos `player_wrap.cc` y `player.py`:

```
swig -shadow -c++ -python player.i
```

El *wrapper* se debe compilar y enlazar con la implementación de la clase en una librería dinámica que se puede importar directamente desde el intérprete.

Listado 22.39: Testeando nuestro nuevo módulo Python

```
1  import player
2  p = player.Player('Carpanta')
3  dir(player)
4  p.printConf()
5  p.position("Goalkeeper")
6  p.printConf()
```

22.2.5. Conclusiones

Realizar un tutorial completo y guiado de la interacción entre C++ y los lenguajes de *scripting* queda fuera del ámbito de este libro. Hemos proporcionado, no obstante, algunos ejemplos sencillos que permiten al lector hacerse una idea de los pasos básicos para una interacción básica entre C++ y un lenguaje de *scripting* de propósito general como es Python.

Se inició esta sección proporcionando los motivos por los que la integración de varios lenguajes de programación en una misma aplicación es una técnica muy útil y ampliamente utilizada en el mundo de los videojuegos. El objetivo final es utilizar el lenguaje mas apropiado para la tarea que estamos desarrollando, lo cual da como resultado una mayor productividad y juegos mas flexibles y extensibles.

A continuación hemos proporcionado algunas directivas básicas de cómo decidir entre lenguajes de scripting y compilados y qué partes son apropiadas para unos lenguajes u otros.

La mayor parte de esta sección se ha dedicado a mostrar cómo podemos integrar C++ y Python de tres maneras posibles:

- El soporte nativo del intérprete de Python es lo mas básico y de más bajo nivel que hay para integrar Python en C++ o viceversa. La documentación del intérprete puede ayudar al lector a profundizar en este API.
- La librería boost nos aporta una visión orientada a objetos y de mas alto nivel para la interacción entre estos lenguajes. Esta librería, o mejor dicho conjunto de librerías, de propósito general nos ayuda en este aspecto particular y nos proporciona otras potentes herramientas de programación en otros ámbitos como hemos visto a lo largo de este curso.
- Por último, hemos introducido la herramienta SWIG que nos puede simplificar de manera extraordinaria la generación de *wrappers* para nuestro código C++ de una forma automática y sin tener que introducir código adicional en nuestro código para interaccionar con Python.

Herramientas y librerías similares a estas están disponibles para otros lenguajes de programación como Lua, prolog, etc.

Capítulo 23
Optimización

Francisco Moya Fernández

Antes de entrar en materia vamos a matizar algunos conceptos. Optimizar hace referencia a obtener el mejor resultado posible. Pero la bondad o maldad del resultado depende fuertemente de los criterios que se pretenden evaluar. Por ejemplo, si queremos hacer un programa lo más pequeño posible el resultado será bastante diferente a si lo que queremos es el programa más rápido posible. Por tanto cuando hablamos de optimización debemos acompañar la frase con el objetivo, con la magnitud que se pretende mejorar hasta el limite de lo posible. Así se habla frecuentemente de optimizar en velocidad u optimizar en tamaño.

La optimización normalmente es un proceso iterativo e incremental. Cada etapa produce un resultado mejor (o por lo menos más fácil de mejorar). A cada una de estas etapas del proceso se les suele denominar también optimizaciones, aunque sería más correcto hablar de etapas del proceso de optimización. Pero además el objetivo de optimización se enmarca en un contexto:

- Las mismas optimizaciones que en una arquitectura concreta generan mejores resultados pueden afectar negativamente al resultado en otras arquitecturas. Por ejemplo, la asignación de variables (o parámetros) a registros en un PowerPC aprovecha el hecho de disponer de un buen número de registros de propósito general. Si se usara el mismo algoritmo para asignar registros en un x86, en el que la mayoría de los registros son de propósito específico, obligaría a introducir multitud de instrucciones adicionales para almacenar temporalmente en la pila.

- Las mismas optimizaciones que permiten mejorar el rendimiento en un procesador pueden perjudicar al rendimiento cuando usamos multiprocesadores o procesadores multi-core. Por ejemplo, el paso por referencia, que permite ahorrar copias innecesarias, también exige utilizar primitivas de sincronización cuando los datos se acceden desde diferentes procesos. Estas primitivas afectan al paralelismo global y los bloqueos pueden superar con mucho el tiempo de copia del objeto.
- Incluso dentro de una misma arquitectura hay optimizaciones que penalizan a determinados procesadores de la misma familia. Por ejemplo en la familia Intel Pentium la forma más eficiente para transferir bloques de memoria era mediante el uso de instrucciones del coprocesador matemático debido al mayor tamaño de dichos registros frente a los de propósito general [71]. Eso ya no aplica para ninguna de las variantes modernas de la familia x86.

En cualquier caso es muy importante tener presente el objetivo global desde el principio, porque las oportunidades de mejora más destacables no están en mano del compilador, sino del programador. Los algoritmos y las estructuras de datos empleados son los que verdaderamente marcan la diferencia, varios órdenes de magnitud mejor que otras alternativas.

El programador de videojuegos siempre tiene que mantener un equilibrio entre dos frases célebres de Donald Knuth[1]:

1. *In established engineering disciplines a 12 % improvement, easily obtained, is never considered marginal and I believe the same viewpoint should prevail in software engineering.* En las disciplinas de ingeniería tradicionales una mejora de un 12 %, fácil de obtener, nunca se considera marginal y pienso que el mismo punto de vista debe prevalecer en la ingeniería de software.
2. *Premature optimization is the root of all evil.* La optimización prematura es la raíz de toda maldad.

Es decir, cuando se está desarrollando un videojuego la optimización no es una prioridad. No debemos ocuparnos de mejorar cuando todavía no sabemos qué debemos mejorar. Está ampliamente documentado que el ser humano es extremadamente malo prediciendo cuellos de botella.

Pero eso no puede justificar la programación descuidada. No es justificable incluir fragmentos de código o algoritmos claramente ineficientes cuando se puede hacer bien desde el principio a un mínimo coste, o incluso a un coste menor.

23.1. Perfilado de programas

Una vez que se dispone de un prototipo o un fragmento funcional del programa podemos determinar los cuellos de botella del programa para intentar mejorarlo. Para ello se suele emplear una técnica conocida como perfilado de software (*software profiling*). El perfilado permite contestar preguntas como:

- ¿Dónde se gasta la mayor parte del tiempo de ejecución? De cara a concentrar los esfuerzos de optimización donde más se notará.
- ¿Cuál es el camino crítico? Para incrementar las prestaciones globales. Por ejemplo, el número de *frames* por segundo.

[1] Ambas frases aparecen prácticamente juntas en la página 268 de [57].

- ¿Cuál es la tasa de fallos de la memoria caché? Con el objetivo de mejorar la localidad de la memoria.

Normalmente recabar este tipo de información implica instrumentar el código añadiendo algunas instrucciones que permiten acumularla en un archivo (o varios) para cada ejecución del programa. La información de perfilado es posteriormente analizada con un programa, el perfilador o *profiler*.

Cada *profiler* implementa el registro de la información de forma diferente. Básicamente se utilizan cuatro técnicas: trazas, muestreo estadístico, puntos de ruptura hardware y contadores hardware. Veamos cada una de ellas en más detalle:

- Cuando el evento de interés corresponde a una operación que requiere un tiempo considerable es posible trazar cada ejecución de la operación sin un impacto significativo en las prestaciones del programa. Ésta es la técnica empleada por el perfilador de Linux *perf* (descrito más adelante) para trazar las operaciones sobre el sistema de archivos, las operaciones de *writeback*, las operaciones de gestión de energía, la recepción y el manejo de interrupciones, las operaciones de planificación de procesos, etc. También es la técnica empleada por utilidades como *strace*, que traza las llamadas al sistema de un proceso.

- Sin embargo, en un programa de tamaño considerable no es posible ejecutar código adicional en todos los eventos de interés (por ejemplo, en todas las llamadas a función). En ese caso se realiza un análisis estadístico. Periódicamente se realiza un muestreo del contador de programa y se analiza en qué función se encuentra. Es más, en lugar de solo observar el valor del contador de programa puede analizar el contenido de la pila para determinar todos marcos de pila activos, es decir, la *call trace*. Con esto es posible determinar el grafo de llamadas y el tiempo estimado destinado a cada función.

- En lugar de instrumentar el código o muestrear de forma estadística, es posible utilizar los mecanismos previstos en los procesadores actuales para facilitar el perfilado. Por ejemplo, una posibilidad es el empleo de puntos de ruptura hardware para detectar cuándo se escribe una posición de memoria, cuándo se escribe, o cuándo se ejecuta la instrucción que contiene. Esta técnica se puede emplear para trazar solo un conjunto limitado de funciones, o para estudiar el patrón de accesos a un objeto. También se emplea en la utilidad *ltrace*, que traza las llamadas a procedimientos de bibliotecas dinámicas desde un proceso determinado.

- Por último los procesadores modernos proporcionan otra funcionalidad especialmente interesante para el perfilado. Disponen de una *Performance Monitoring Unit* que controla un conjunto de registros especiales denominados *performance counters*. Estos registros son capaces de contar determinados eventos, tales como ciclos de la CPU, ciclos de bus, instrucciones, referencias a la cache, fallos de la memoria caché, saltos o fallos en la predicción de saltos. Estos registros pueden ser utilizados en *profilers* tales como *perf* para realizar mediciones muy precisas.

Es importante conocer cuándo se emplea cada una de estas técnicas para poder interpretar con precisión los datos del perfilado. Así, por ejemplo, las técnicas basadas en muestreo de la traza de llamadas debe entenderse en un contexto estadístico. Valores bajos en los contadores de llamadas no tienen significado absoluto, sino en relación a otros contadores. Es muy posible que tengamos que ejecutar el mismo fragmento de código múltiples veces para eliminar cualquier sesgo estadístico.

Para cualquier análisis que requiera examinar la pila (perfilado de la traza de llamadas, o del grafo de llamadas, o simplemente la depuración interactiva), se asume el convenio de que un registro contiene la dirección del marco de pila actual (*frame pointer*) y al principio del marco de pila actual se almacena una copia del *frame pointer* anterior a la llamada actual.
Sin embargo los compiladores actuales pueden generar código perfectamente funcional sin necesidad de *frame pointer*. Es importante compilar los programas evitando la opción `-fomit-frame-pointer` o incluso explícitamente indicando `-fno-omit-frame-pointer` durante el desarrollo para que estos análisis funcionen correctamente.

23.1.1. El perfilador de Linux *perf*

El subsistema *Linux Performance Counters* proporciona una abstracción de los *performance counters* disponibles en los procesadores modernos. Independientemente del hardware subyacente Linux ofrece una serie de contadores de 64 bits virtualizados por CPU o por tarea y combinado con un sistema de traza de eventos de otro tipo (eventos software, trazas). Es más sencillo de lo que parece, veamos algún ejemplo.

En las distribuciones más actuales, la herramienta *perf* está incluida en el paquete `linux-base`. Pero se trata de un simple envoltorio para ejecutar la correspondiente al kernel que se está ejecutando. El ejecutable real se encuentra en el paquete `linux-tools-X.Y` donde X.Y hace referencia a la versión del kernel empleada. Por ejemplo, `linux-tools-3.2` o `linux-tools-3.8`.

Por tanto para instalar la herramienta deberemos ejecutar:

```
$ sudo apt-get install linux-base linux-tools-3.2
```

A continuación conviene configurar el kernel para que permita a los usuarios normales recabar estadísticas de todo tipo. Esto no debe hacerse con carácter general, sino solo en las computadoras empleadas en el desarrollo, puesto que también facilita la obtención de información para realizar un ataque.

```
$ sudo sh -c "echo -1 > /proc/sys/kernel/perf_event_paranoid"
```

Ahora ya como usuarios normales podemos perfilar cualquier ejecutable, e incluso procesos en ejecución. Tal vez la primera tarea que se debe realizar para perfilar con `perf` es obtener la lista de eventos que puede contabilizar. Esta lista es dependiente de la arquitectura del procesador y de las opciones de compilación del kernel.

```
$ perf list

List of pre-defined events (to be used in -e):
  cpu-cycles OR cycles                                [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend     [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend       [Hardware event]
  instructions                                        [Hardware event]
  cache-references                                    [Hardware event]
  cache-misses                                        [Hardware event]
  branch-instructions OR branches                     [Hardware event]
  branch-misses                                       [Hardware event]
  bus-cycles                                          [Hardware event]

  cpu-clock                                           [Software event]
  task-clock                                          [Software event]
```

```
page-faults OR faults                                     [Software event]
minor-faults                                              [Software event]
major-faults                                              [Software event]
context-switches OR cs                                    [Software event]
cpu-migrations OR migrations                              [Software event]
alignment-faults                                          [Software event]
emulation-faults                                          [Software event]

L1-dcache-loads                                   [Hardware cache event]
L1-dcache-load-misses                             [Hardware cache event]
L1-dcache-stores                                  [Hardware cache event]
L1-dcache-store-misses                            [Hardware cache event]
L1-dcache-prefetches                              [Hardware cache event]
L1-dcache-prefetch-misses                         [Hardware cache event]
L1-icache-loads                                   [Hardware cache event]
L1-icache-load-misses                             [Hardware cache event]
L1-icache-prefetches                              [Hardware cache event]
L1-icache-prefetch-misses                         [Hardware cache event]
LLC-loads                                         [Hardware cache event]
LLC-load-misses                                   [Hardware cache event]
LLC-stores                                        [Hardware cache event]
LLC-store-misses                                  [Hardware cache event]
LLC-prefetches                                    [Hardware cache event]
LLC-prefetch-misses                               [Hardware cache event]
dTLB-loads                                        [Hardware cache event]
dTLB-load-misses                                  [Hardware cache event]
dTLB-stores                                       [Hardware cache event]
dTLB-store-misses                                 [Hardware cache event]
dTLB-prefetches                                   [Hardware cache event]
dTLB-prefetch-misses                              [Hardware cache event]
iTLB-loads                                        [Hardware cache event]
iTLB-load-misses                                  [Hardware cache event]
branch-loads                                      [Hardware cache event]
branch-load-misses                                [Hardware cache event]
node-loads                                        [Hardware cache event]
node-load-misses                                  [Hardware cache event]
node-stores                                       [Hardware cache event]
node-store-misses                                 [Hardware cache event]
node-prefetches                                   [Hardware cache event]
node-prefetch-misses                              [Hardware cache event]

rNNN (...)                                  [Raw hardware event descriptor]

mem:<addr>[:access]                                   [Hardware breakpoint]

i915:i915_gem_object_create                              [Tracepoint event]
i915:i915_gem_object_bind                                [Tracepoint event]
i915:i915_gem_object_unbind                              [Tracepoint event]
...
sched:sched_wakeup                                       [Tracepoint event]
sched:sched_wakeup_new                                   [Tracepoint event]
sched:sched_switch                                       [Tracepoint event]
...
```

En la lista de eventos podemos apreciar seis tipos diferentes.

- *Software event*. Son simples contadores del kernel. Entre otros permite contar cambios de contexto o fallos de página.
- *Hardware event*. Este evento se refiere a los contadores incluidos en las PMU (Performance Monitoring Units) de los procesadores modernos. Permite contar ciclos, intrucciones ejecutadas, fallos de caché. Algunos de estos contadores se ofrecen de forma unificada como contadores de 64 bits, de tal forma que oculta los detalles de la PMU subyacente. Pero en general su número y tipo dependerá del modelo de rocesador donde se ejecuta.

- *Hardware cache event*. Dentro del subsistema de memoria las PMU modernas[2] permiten extraer estadísticas detalladas de las memorias caché de primer nivel de último nivel o del TLB (Translation Lookaside Buffer). Nuevamente se trata de contadores que dependen fuertemente del modelo de procesador sobre el que se ejecuta.
- *Hardware breakpoint*. Los puntos de ruptura hardware permiten detener la ejecución del programa cuando el procesador intenta leer, escribir o ejecutar el contenido de una determinada posición de memoria. Esto nos permite monitorizar detalladamente objetos de interés, o trazar la ejecución de instrucciones concretas.
- *Tracepoint event*. En este caso se trata de trazas registradas con la infraestructura *ftrace* de Linux. Se trata de una infraestructura extremadamente flexible para trazar todo tipo de eventos en el kernel o en cualquier módulo del kernel. Esto incluye eventos de la GPU, de los sistemas de archivos o del propio *scheduler*.
- *Raw hardware event*. En el caso de que `perf` no incluya todavía un nombre simbólico para un contador concreto de una PMU actual se puede emplear el código hexadecimal correspondiente, de acuerdo al manual del fabricante.

23.1.2. Obteniendo ayuda

La primera suborden de `perf` que debe dominarse es `help`, que se emplea para obtener ayuda. La ejecución de `perf help` sin más nos muestra todas las órdenes disponibles. Las más utilizadas son `perf stat`, `perf record`, `perf report` y `perf annotate`.

Cada una de estas órdenes tienen ayuda específica que puede obtenerse con `perf help` *suborden*.

23.1.3. Estadísticas y registro de eventos

La operación más sencilla que se puede hacer con `perf` es contar eventos. Eso puede realizarse con la suborden `perf stat`:

```
$ perf stat glxgears

 Performance counter stats for 'glxgears':

  80,416861 task-clock              #    0,069 CPUs utilized
       171 context-switches         #    0,002 M/sec
        71 CPU-migrations           #    0,001 M/sec
     10732 page-faults              #    0,133 M/sec
 109061681 cycles                   #    1,356 GHz                      [86,41%]
  75057377 stalled-cycles-frontend  #   68,82% frontend cycles idle     [85,21%]
  58498153 stalled-cycles-backend   #   53,64% backend  cycles idle     [62,34%]
  68356682 instructions             #    0,63  insns per cycle
                                    #    1,10  stalled cycles per insn [80,66%]
  14463080 branches                 #  179,851 M/sec                    [86,78%]
    391522 branch-misses            #    2,71% of all branches          [80,19%]

1,158777481 seconds time elapsed
```

[2]Los eventos de las PMU se documentan en los manuales de los fabricantes. Por ejemplo, los contadores de la arquitectura Intel 64 e IA32 se documentan en el apéndice A de [48] disponible en `http://www.intel.com/Assets/PDF/manual/253669.pdf` y los de los procesadores AMD64 en [5] disponible en `http://support.amd.com/us/Processor_TechDocs/31116.pdf`

Basta indicar el ejecutable a continuación de `perf stat`. Por defecto muestra un conjunto de métricas comunes, que incluye eventos hardware (como los ciclos o las instrucciones), eventos software (como los cambios de contexto), y métricas derivadas a la derecha (como el número de instrucciones por ciclo).

Puede utilizarse `perf` para medir un tipo de eventos concreto empleando la opción `-e`:

```
$ perf stat -e cycles,instructions precompute_landscape

 Performance counter stats for 'precompute_landscape':

      4473759 cycles           #  0,000 GHz
      3847463 instructions     #  0,86  insns per cycle

  0,004595748 seconds time elapsed
```

Y podemos dividir entre los eventos que ocurren en espacio de usuario y los que ocurren en espacio del kernel.

```
$ perf stat -e cycles:u,cycles:k precompute_landscape

 Performance counter stats for 'precompute_landscape':

           1827737 cycles:u     #  0,000 GHz
           2612202 cycles:k     #  0,000 GHz

       0,005022949 seconds time elapsed
```

Todos los eventos hardware aceptan los modificadores `u` para filtrar solo los que ocurren en espacio de usuario, `k` para filtrar los que ocurren en espacio del kernel y `uk` para contabilizar ambos de forma explícita. Hay otros modificadores disponibles, incluso alguno dependiente del procesador en el que se ejecuta.

23.1.4. Multiplexación y escalado

Las PMU tienen dos tipos de contadores: los contadores fijos, que cuentan un único tipo de evento, y los contadores genéricos, que pueden configurarse para contar cualquier evento hardware. Cuando el usuario solicita más eventos de los que físicamente se pueden contar con los contadores implementados el sistema de perfilado multiplexa los contadores disponibles. Esto hace que parte del tiempo se estén contando unos eventos y parte del tiempo se están contando otros eventos distintos.

Posteriormente el propio sistema escala los valores calculados en proporción al tiempo que se ha contado el evento respecto al tiempo total. Es muy fácil de ver el efecto con un ejemplo. El computador sobre el que se escriben estas líneas dispone de un procesador Intel Core i5. Estos procesadores tienen 4 contadores genéricos[3].

Vamos a ver qué pasa cuando se piden 4 eventos idénticos:

```
$ perf stat -e cycles,cycles,cycles,cycles render_frame

 Performance counter stats for 'render_frame':

   803261796 cycles    #  0,000 GHz
   803261796 cycles    #  0,000 GHz
   803261796 cycles    #  0,000 GHz
   803261799 cycles    #  0,000 GHz
```

[3]Lo más normal es disponer de dos o cuatro contadores genéricos y otros tantos específicos. Realiza la misma prueba en tu ordenador para comprobar cuántos contadores genéricos tiene.

```
 0,306640126 seconds time elapsed
```

Puede verse que la precisión es absoluta, los cuatro contadores han contado prácticamente la misma cantidad de ciclos. En cambio, veamos qué pasa cuando se solicitan 5 eventos idénticos:

```
$ perf stat -e cycles,cycles,cycles,cycles,cycles render_frame

 Performance counter stats for 'render_frame':

   801863997 cycles    #  0,000 GHz   [79,06%]
   801685466 cycles    #  0,000 GHz   [80,14%]
   792515645 cycles    #  0,000 GHz   [80,37%]
   792876560 cycles    #  0,000 GHz   [80,37%]
   793921257 cycles    #  0,000 GHz   [80,08%]

 0,306024538 seconds time elapsed
```

Los valores son significativamente diferentes, pero los porcentajes entre corchetes nos previenen de que se ha realizado un escalado. Por ejemplo, el primer contador ha estado contabilizando ciclos durante el 79,06 % del tiempo. El valor obtenido en el contador se ha escalado dividiendo por 0,7906 para obtener el valor mostrado.

En este caso los contadores nos dan una aproximación, no un valor completamente fiable. Nos vale para evaluar mejoras en porcentajes significativos, pero no mejoras de un 1 %, porque como vemos el escalado ya introduce un error de esa magnitud. Además en algunas mediciones el resultado dependerá del momento concreto en que se evalúen o de la carga del sistema en el momento de la medida. Para suavizar todos estos efectos estadísticos se puede ejecutar varias veces empleando la opción `-r`.

```
$ perf stat -r 10 -e cycles,cycles,cycles,cycles,cycles render_frame

 Performance counter stats for 'render_frame'' (10 runs):

   803926738 cycles  # 0,000 GHz  ( +-  0,15% ) [79,42%]
   804290331 cycles  # 0,000 GHz  ( +-  0,14% ) [79,66%]
   802303057 cycles  # 0,000 GHz  ( +-  0,17% ) [80,21%]
   797518018 cycles  # 0,000 GHz  ( +-  0,11% ) [80,59%]
   799832288 cycles  # 0,000 GHz  ( +-  0,19% ) [80,15%]

 0,310143008 seconds time elapsed ( +-  0,39% )
```

Entre paréntesis se muestra la variación entre ejecuciones.

23.1.5. Métricas por hilo, por proceso o por CPU

Es posible contabilizar los eventos solo en un hilo, o en todos los hilos de un proceso, o en todos los procesos de una CPU, o de un conjunto de ellas. Por defecto `perf` contabiliza eventos del hilo principal incluyendo todos los subprocesos, creados con `fork()`, o hilos, creados con `pthread_create()`, lanzados durante la ejecución. Este comportamiento se implementa con un mecanismo de herencia de contadores que puede desactivarse con la opción `-i` de `perf stat`.

Alternativamente se puede recolectar datos de un conjunto de procesadores en lugar de un proceso concreto. Este modo se activa con la opción `-a` y opcionalmente complementado con la opción `-C`. Al utilizar la opción `-a` se activa la recolección de datos por CPU, pero por defecto se agregan todos los contadores de todas las CPU

(recolección de datos a nivel de sistema). Con la opción `-C` podemos seleccionar la CPU o conjunto de CPUs de los que se recaban estadísticas. Por ejemplo, para recolectar el número de fallos de página en espacio de usuario de las CPUs 0 y 2 durante 5 segundos:

```
$ perf stat -a -e faults -C 0,2 sleep 5

 Performance counter stats for 'sleep 5':

               233 faults

       5,001227158 seconds time elapsed
```

Nótese que utilizamos la orden `sleep` para no consumir ciclos y de esta forma no influir en la medida.

23.1.6. Muestreo de eventos

Además de contar eventos, `perf` puede realizar un muestreo similar a otros *profilers*. En este caso `perf record` recolecta datos en un archivo llamado `perf.data` que posteriormente puede analizarse con `perf report` o `perf annotate`.

El periodo de muestreo se especifica en número de eventos. Si el evento que se utiliza para el muestreo es `cycles` (es así por defecto) entonces el periodo tiene relación directa con el tiempo, pero en el caso general no tiene por qué. Incluso en el caso por defecto la relación con el tiempo no es lineal, en caso de que el procesador tenga activos modos de escalado de frecuencia.

Por defecto `perf record` registra 1000 muestras por segundo y ajusta dinámicamente el periodo para que mantenga esta tasa. El usuario puede establecer una frecuencia de muestreo utilizando la opción `-F` o puede establecer un periodo fijo con la opción `-c`.

A diferencia de otros *profilers*, `perf record` puede recoger estadísticas a nivel del sistema completo o de un conjunto de CPUs concreto empleando las opciones `-a` y `-C` que ya hemos visto al explicar `perf stat`.

Es especialmente interesante el muestreo de la traza de llamada empleando la opción `-g`, aunque para que esta característica muestre resultados fiables es necesario mantener el convenio de *marcos de llamada* y no compilar mediante la opción `-fomit-frame-pointer`.

Para mostrar los resultados almacenados en `perf.data` puede emplearse `perf report`. Por ejemplo, a continuación recolectaremos datos de la actividad del sistema durante 5 segundos y mostraremos el resultado del perfilado.

```
$ perf record -a sleep 10
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.426 MB perf.data (~18612 samples)
   ]
$ perf report
```

El muestreo permite analizar qué funciones se llevan la mayor parte del tiempo de ejecución y, en caso de muestrear también la traza de llamada permite identificar de forma rápida los cuellos de botella. No se trata de encontrar la función que más tiempo se lleva, sino de identificar funciones que merezca la pena optimizar. No tiene sentido optimizar una función que se lleva el 0,01 % del tiempo de ejecución, porque simplemente no se notaría.

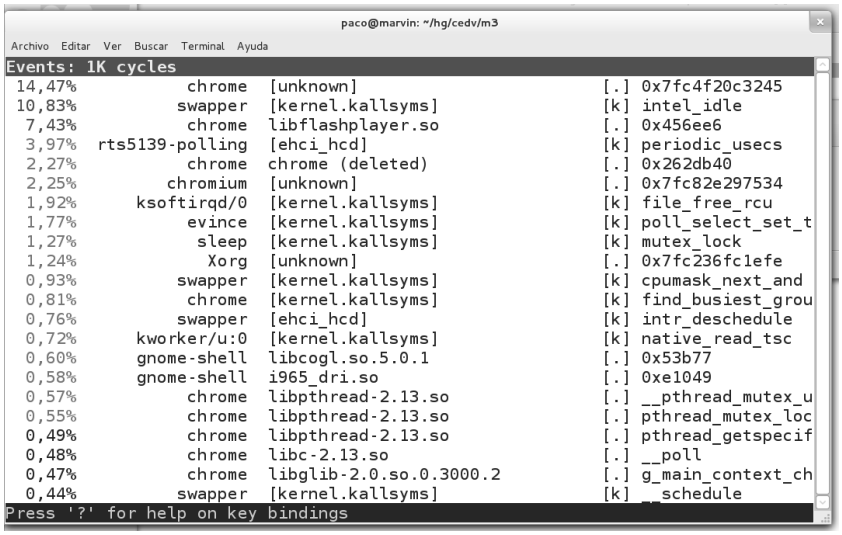


Figura 23.1: Interfaz textual de `perf report`.

Una vez identificada la función o las funciones susceptibles de mejora podemos analizarlas en mayor detalle, incluso a nivel del código ensamblador empleando `perf annotate` *símbolo*. También desde la interfaz de texto es posible examinar el código anotado seleccionando el símbolo y pulsando la tecla [a].

Para poder utilizar las características de anotación del código de los perfiladores es necesario compilar el programa con información de depuración.

Para ilustrar la mecánica veremos un caso real. Ingo Molnar, uno de los principales desarrolladores de la infraestructura de perfilado de Linux, tiene multitud de mensajes en diversos foros sobre optimizaciones concretas que fueron primero identificadas mediante el uso de `perf`. Uno de ellos[4] describe una optimización significativa de `git`, el sistema de control de versiones.

En primer lugar realiza una fase de análisis de una operación concreta que revela un dato intranquilizador. Al utilizar la operación de compactación `git gc` descubre un número elevado de ciclos de estancamiento (*stalled cycles*[5]):

```
$ perf record -e stalled-cycles -F 10000 ./git gc
$ perf report --stdio

# Events: 26K stalled-cycles
#
# Overhead  Command     Shared Object           Symbol
# ........  ..........  ......................  ........................
#
    26.07%         git  git                     [.] lookup_object
```

[4] `http://thread.gmane.org/gmane.comp.version-control.git/172286`

[5] En las versiones actuales de `perf` habría que usar `stalled-cycles-frontend` en lugar de `stalled-cycles` pero mantenemos el texto del caso de uso original para no confundir al lector.

```
    10.22 %         git  libz.so.1.2.5          [.] 0xc43a
     7.08 %         git  libz.so.1.2.5          [.] inflate
     6.63 %         git  git                    [.] find_pack_entry_one
     5.37 %         git  [kernel.kallsyms]      [k] do_raw_spin_lock
     4.03 %         git  git                    [.] lookup_blob
     3.09 %         git  libc-2.13.90.so        [.] __strlen_sse42
     2.81 %         git  libc-2.13.90.so        [.] __memcpy_ssse3_back
```

Ingo descubre que la función `find_pack_entry_one()` se lleva un porcentaje significativo de los ciclos de estancamiento. Por tanto examina el contenido de esa función con `perf annotate`. Para poder extraer todo el beneficio de esta orden es interesante compilar el programa con información de depuración.

```
$ perf annotate find_pack_entry_one
```

```
 Percent | Source code & Disassembly of git
------------------------------------------
         :
     ...
         :           int cmp = hashcmp(index + mi * stride, sha1);
    0.90 : 4b9264:  89 ee              mov    %ebp,%esi
    0.45 : 4b9266:  41 0f af f2        imul   %r10d,%esi
    2.86 : 4b926a:  4c 01 de           add    %r11,%rsi
   53.34 : 4b926d:  f3 a6              repz cmpsb %es:(%rdi),%ds:(%rsi)
   14.37 : 4b926f:  0f 92 c0           setb   %al
    5.78 : 4b9272:  41 0f 97 c4        seta   %r12b
    1.52 : 4b9276:  41 28 c4           sub    %al,%r12b
```

La mayoría de la sobrecarga está en la función `hashcmp()` que usa `memcmp()`, pero esta última se expande como instrucciones ensamblador por el propio compilador.

Ingo Molnar estudia el caso concreto. La función `hashcmp()` compara *hashes*, y por eso se utiliza `memcmp()`, pero si no coinciden el primer byte diferirá en el 99 % de los casos. Por tanto modifica el programa para escribir la comparación manualmente, evitando entrar en la comparación para la mayor parte de los casos.

El resultado es realmente sorprendente. Antes de la optimización obtuvo estos números:

```
$ perf stat --sync --repeat 10 ./git gc

Performance counter stats for './git gc' (10 runs):

  2771.119892 task-clock       #    0.863 CPUs utilized           ( +-  0.16 % )
        1,813 context-switches #    0.001 M/sec                   ( +-  3.06 % )
          167 CPU-migrations   #    0.000 M/sec                   ( +-  2.92 % )
       39,210 page-faults      #    0.014 M/sec                   ( +-  0.26 % )
8,828,405,654 cycles           #    3.186 GHz                     ( +-  0.13 % )
2,102,083,909 stalled-cycles   #   23.81 % of all cycles are idle  ( +-  0.52 % )
8,821,931,740 instructions     #    1.00  insns per cycle
                               #    0.24  stalled cycles per insn ( +-  0.04 % )
1,750,408,175 branches         #  631.661 M/sec                   ( +-  0.04 % )
   74,612,120 branch-misses    #    4.26 % of all branches        ( +-  0.07 % )

 3.211098537  seconds time elapsed  ( +-  1.52 % )
```

La opción `-sync` hace que se ejecute una llamada `sync()` (vuelca los buffers pendientes de escritura de los sistemas de archivos) antes de cada ejecución para reducir el ruido en el tiempo transcurrido.

Después de la optimización el resultado es:

```
$ perf stat --sync --repeat 10 ./git gc

Performance counter stats for './git gc' (10 runs):

   2349.498022 task-clock       #    0.807 CPUs utilized           ( +-  0.15 % )
```

```
         1,842 context-switches #    0.001 M/sec                    ( +-  2.50% )
           164 CPU-migrations   #    0.000 M/sec                    ( +-  3.67% )
        39,350 page-faults      #    0.017 M/sec                    ( +-  0.06% )
 7,484,317,230 cycles           #    3.185 GHz                      ( +-  0.15% )
 1,577,673,341 stalled-cycles   #   21.08% of all cycles are idle   ( +-  0.67% )
11,067,826,786 instructions     #    1.48  insns per cycle
                                #    0.14  stalled cycles per insn ( +-  0.02% )
 2,489,157,909 branches         # 1059.442 M/sec                    ( +-  0.02% )
    59,384,019 branch-misses    #    2.39% of all branches         ( +-  0.22% )

    2.910829134  seconds time elapsed  ( +-  1.39% )
```

La misma operación se aceleró en un 18 %. Se han eliminado el 33 % de los ciclos de estancamiento y la mayoría de ellos se han traducido en ahorro efectivo de ciclos totales y con ello en mejoras de velocidad.

Este ejemplo deja claro que las instrucciones ensamblador que emite el compilador para optimizar `memcmp()` no son óptimas para comparaciones pequeñas. La instrucción `repz cmpsb` requiere un tiempo de *setup* considerable durante el cual la CPU no hace nada más.

Otro efecto interesante que observa Ingo Molnar sobre esta optimización es que también mejora la predicción de saltos. Midiendo el evento `branch-misses` obtiene los siguientes resultados:

	branch-misses	**% del total**
Antes	74,612,120	4.26 % ($\pm$ 0.07 %)
Después	59,384,019	2.39 % ($\pm$ 0.22 %)

Cuadro 23.1: Mejora en predicción de saltos

Por alguna razón el bucle abierto es más sencillo de predecir por parte de la CPU por lo que produce menos errores de predicción.

No obstante es importante entender que estas optimizaciones corresponden a problemas en otros puntos (compilador que genera código subóptimo, y arquitectura que privilegia un estilo frente a otro). Por tanto se trata de optimizaciones con fecha de caducidad. Cuando se utilice una versión más reciente de GCC u otro compilador más agresivo en las optimizaciones esta optimización no tendrá sentido.

Un caso célebre similar fue la optimización del recorrido de listas en el kernel Linux[6]. Las listas son estructuras muy poco adecuadas para la memoria caché. Al no tener los elementos contiguos generan innumerables fallos de caché. Mientras se produce un fallo de caché el procesador está parcialmente parado puesto que necesita el dato de la memoria para operar. Por esta razón en Linux se empleó una optimización denominada *prefetching*. Antes de operar con un elemento se accede al siguiente. De esta forma mientras está operando con el elemento es posible ir transfiriendo los datos de la memoria a la caché.

Desgraciadamente los procesadores modernos incorporan sus propias unidades de *prefetch* que realizan un trabajo mucho mejor que el manual, puesto que no interfiere con el TLB. El propio Ingo Molnar reporta que esta optimización estaba realmente causando un impacto de 0,5 %.

La lección que debemos aprender es que nunca se debe optimizar sin medir, que las optimizaciones dependen del entorno de ejecución, y que si el entorno de ejecución varía las optimizaciones deben re-evaluarse.

[6] https://lwn.net/Articles/444336/

Paquete	Herramienta	Descripción
Valgrind Callgrind[7]	`kCacheGrind`	Excelentes capacidades de representación gráfica.
Google Performance Tools[8]	`google-pprof`	Permite perfilado de CPU y de memoria dinámica. Permite salida en formato *callgrind* para poder analizar con `kCacheGrind`.
GNU Profiler[9]	`gprof`	Es una herramienta estándar pero ha ido perdiendo su utilidad conforme fueron surgiendo los perfiladores basados en PMU.
nVidia Visual Profiler	`nvvp`	Es específico para GPUs nVidia.
AMD APP Profiler	`sprofile`	Es específico para GPUs AMD/ATI Radeon.

Cuadro 23.2: Herramientas de perfilado en GNU/Linux.

23.1.7. Otras opciones de `perf`

Puede resultar útil también la posibilidad de contabilizar procesos o hilos que ya están en ejecución (opciones `-p` y `-t` respectivamente). A pesar de usar cualquiera de estas opciones se puede especificar una orden para limitar el tiempo de medición. En caso contrario mediría el proceso o hilo hasta su terminación.

También es posible generar gráficos de líneas temporales. Para ello es necesario utilizar la suborden `perf timechart record` para registrar los eventos de forma similar a como se hacía con `perf record` y posteriormente emplear `perf timechart` para generar el archivo `output.svg`. Este archivo puede editarse o convertirse a PDF (Portable Document Format) con *inkscape*. El problema es que el tiempo de captura debe ser reducido o de lo contrario el archivo SVG se volverá inmanejable. No obstante es muy útil para detectar problemas de bloqueo excesivo. Por ejemplo, los datos de la figura 23.2 se grabaron con `perf timechart record -a sleep 1`.

Por último conviene citar la suborden `perf top` que permite monitorizar en tiempo real el sistema para analizar quién está generando más eventos.

23.1.8. Otros perfiladores

La tabla 23.2 muestra una colección de herramientas de perfilado disponibles en entornos GNU y GNU/Linux.

La mayoría de los perfiladores requieren compilar el programa de una manera especial. El más extendido y portable es GNU Profiler, incluido dentro de `binutils`, que es directamente soportado por el compilador de GNU. Si se compilan y se montan los programas con la opción `-pg` el programa quedará instrumentado para perfilado.

Todas las ejecuciones del programa generan un archivo `gmon.out` con la información recolectada, que puede examinarse con `gprof`. GNU Profiler utiliza muestreo estadístico sin ayuda de PMU. Esto lo hace muy portable pero notablemente impreciso.

Google Performance Tools aporta un conjunto de bibliotecas para perfilado de memoria dinámica o del procesador con apoyo de PMU. Por ejemplo, el perfilado de programas puede realizarse con la biblioteca `libprofiler.so`. Esta biblioteca puede ser cargada utilizando la variable de entorno `LD_PRELOAD` y activada mediante la definición de la variable de entorno `CPUPROFILE`. Por ejemplo:

```
$ LD_PRELOAD=/usr/lib/libprofiler.so.0 CPUPROFILE=prof.data \
  ./light-model-test
```

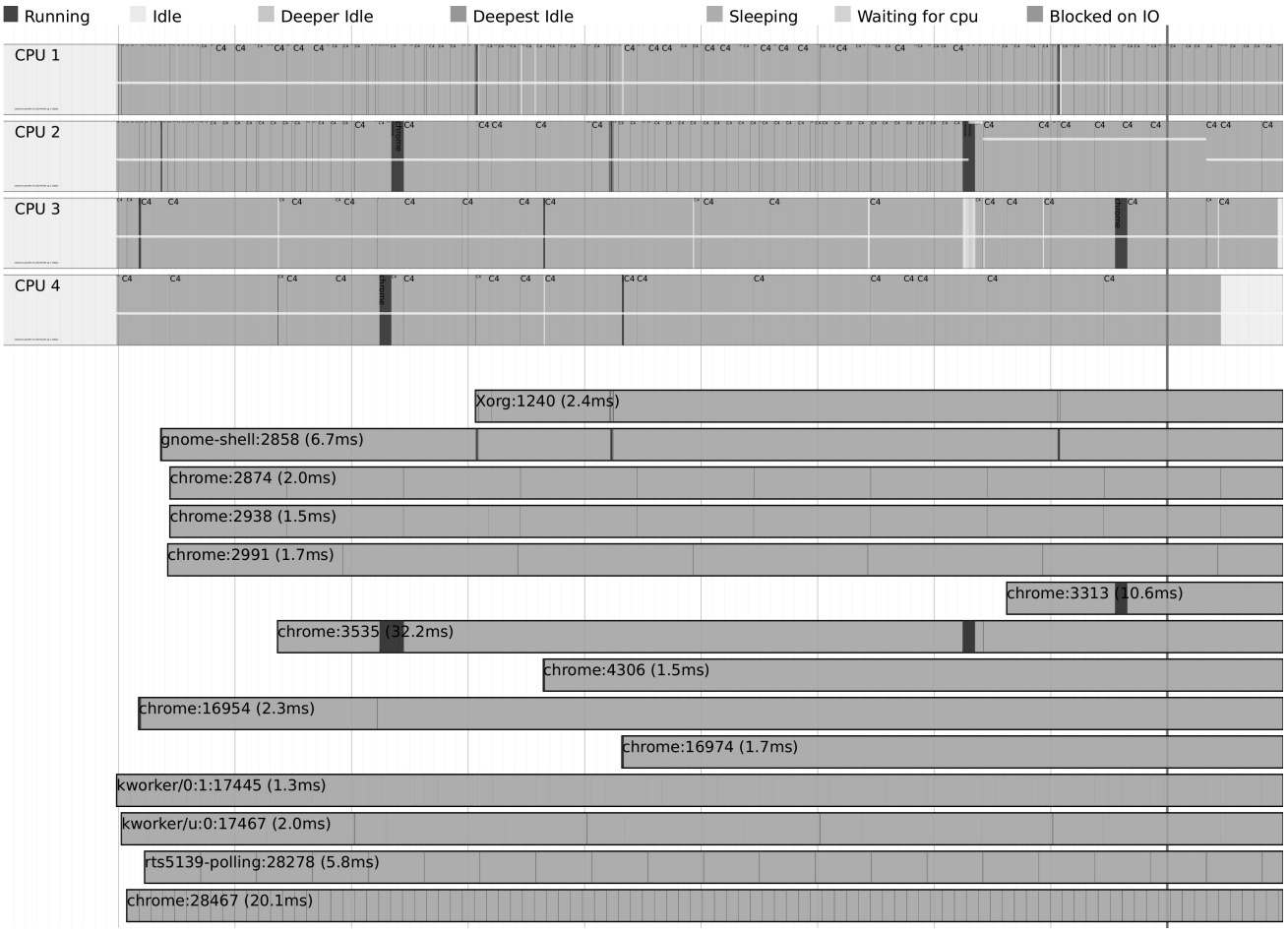


Figura 23.2: Ejemplo de `perf timechart`.

Esto genera el archivo `prof.data` con los datos de perfilado, que luego pueden examinarse con `google-pprof`. Entre otras capacidades permite representación gráfica del grafo de llamadas o compatibilidad con el formato de `kcachegrind`.

Una característica interesante de Google Performance Tools es la capacidad de realizar el perfilado solo para una sección concreta del código. Para ello, en lugar de definir la variable `CPUPROFILE` basta incluir en el código llamadas a las funciones `ProfilerStart()` y `ProfilerStop()`.

Para un desarrollador de videojuegos es destacable la aparición de perfiladores específicos para GPUs. Las propias GPUs tienen una PMU (*Performance Monitoring Unit*) que permite recabar información de contadores específicos. De momento en el mundo del software libre han emergido nVidia Visual Profiler, AMD APP Profiler y extensiones de Intel a `perf` para utilizar los contadores de la GPU (`perf gpu`). Probablemente en un futuro cercano veremos estas extensiones incorporadas en la distribución oficial de `linux-tools`.

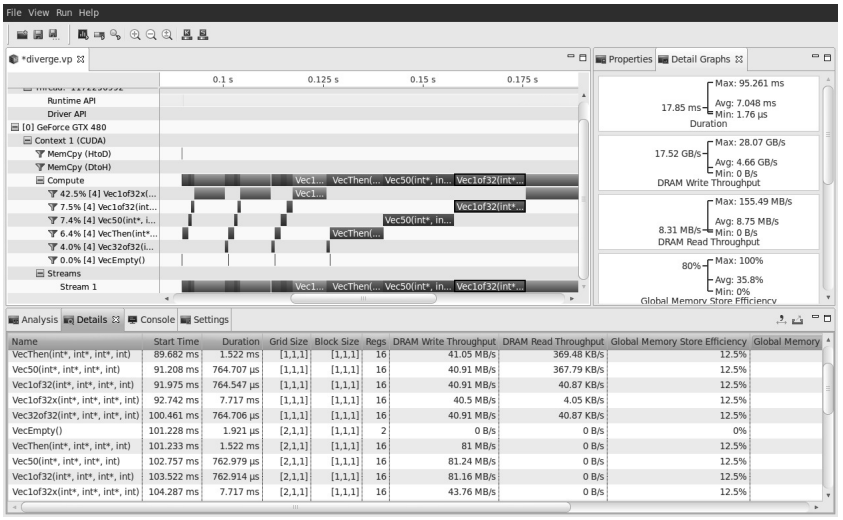
GPU Profilers

De momento solo nVidia proporciona un profiler con capacidades gráficas sobre GNU/Linux. AMD APP Profiler funciona en GNU/Linux pero no con interfaz gráfica.

23.2. Optimizaciones del compilador

Los compiladores modernos implementan un enorme abanico de optimizaciones. Con frecuencia son tan eficientes como el código ensamblador manualmente programado. Por esta razón es cada vez más raro encontrar fragmentos de código ensamblador en programas bien optimizados.

El lenguaje C++, y su ancestro C son considerados como lenguajes de programación de sistemas. Esto se debe a que permiten acceso a características de muy bajo nivel, hasta el punto de que algunos autores lo consideran un *ensamblador portable*. Los punteros no dejan de ser una forma de expresar direccionamiento indirecto, así como el operador de indexación no deja de ser una expresión de los modos de direccionamiento relativo.



Name	Start Time	Duration	Grid Size	Block Size	Regs	DRAM Write Throughput	DRAM Read Throughput	Global Memory Store Efficiency	Global Memory
VecThen(int*, int*, int*, int)	89.682 ms	1.522 ms	[1,1,1]	[1,1,1]	16	41.05 MB/s	369.48 KB/s	12.5%	
Vec50(int*, int*, int*, int)	91.208 ms	764.707 µs	[1,1,1]	[1,1,1]	16	40.91 MB/s	367.79 KB/s	12.5%	
Vec1of32(int*, int*, int*, int)	91.975 ms	764.547 µs	[1,1,1]	[1,1,1]	16	40.91 MB/s	40.87 KB/s	12.5%	
Vec1of32x(int*, int*, int*, int)	92.742 ms	7.717 ms	[1,1,1]	[1,1,1]	16	40.5 MB/s	4.05 KB/s	12.5%	
Vec32of32(int*, int*, int*, int)	100.461 ms	764.706 µs	[1,1,1]	[1,1,1]	16	40.91 MB/s	40.87 KB/s	12.5%	
VecEmpty()	101.228 ms	1.921 µs	[2,1,1]	[1,1,1]	2	0 B/s	0 B/s	0%	
VecThen(int*, int*, int*, int)	101.233 ms	1.522 ms	[2,1,1]	[1,1,1]	16	81 MB/s	0 B/s	12.5%	
Vec50(int*, int*, int*, int)	102.757 ms	762.979 µs	[2,1,1]	[1,1,1]	16	81.24 MB/s	0 B/s	12.5%	
Vec1of32(int*, int*, int*, int)	103.522 ms	762.914 µs	[2,1,1]	[1,1,1]	16	81.16 MB/s	0 B/s	12.5%	
Vec1of32x(int*, int*, int*, int)	104.287 ms	7.717 ms	[2,1,1]	[1,1,1]	16	43.76 MB/s	0 B/s	12.5%	

Figura 23.3: Aspecto de la interfaz de nVidia Visual Profiler.

C fue diseñado con el objetivo inicial de programar un sistema operativo. Por este motivo, desde las primeras versiones incorpora características de muy bajo nivel que permite dirigir al compilador para generar código más eficiente. Variables registro, funciones en línea, paso por referencia, o plantillas son algunas de las características que nos permiten indicar al compilador cuándo debe esforzarse en buscar la opción más rápida. Sin embargo, la mayoría de las construcciones son simplemente indicaciones o sugerencias, que el compilador puede ignorar libremente si encuentra una solución mejor. En la actualidad tenemos compiladores libres maduros con capacidades comparables a los mejores compiladores comerciales, por lo que frecuentemente las indicaciones del programador son ignoradas.

23.2.1. Variables registro

Los más viejos del lugar recordarán un modificador opcional para las variables denominado `register`. Este modificador indica al compilador que se trata de una variable especialmente crítica, por lo que sugiere almacenarla en un registro del procesador. Era frecuente ver código como éste:

Listado 23.1: Utilización arcaica de `register` para sumar los 1000 primeros números naturales.

```
1 register unsigned i, sum = 0;
2 for (i=1; i<1000; ++i)
3   sum += i;
```

Esta palabra clave está en desuso porque los algoritmos de asignación de registros actuales son mucho mejores que la intuición humana. Pero además, aunque se utilizara, sería totalmente ignorada por el compilador. La mera aparición de `register` en un programa debe ser considerada como un *bug*, porque engaña al lector del programa haciéndole creer que dicha variable será asignada a un registro, cuando ese aspecto está fuera del control del programador.

23.2.2. Código estático y funciones *inline*

Ya se ha comentado el uso del modificador `inline` en el módulo 1. Sirve para indicar al compilador que debe replicar el código de dicha función cada vez que aparezca una llamada. Si no se hiciera generaría código independiente para la función, al que salta mediante una instrucción de llamada a subrutina. Sin embargo no siempre es posible la sustitución en línea del código y además el compilador es libre de hacer sustitución en línea de funciones aunque no estén marcadas como `inline`. Veamos un ejemplo:

Listado 23.2: Ejemplo sencillo de función apropiada para la expansión en línea.

```
1  int sum(int* a, unsigned size)
2  {
3      int ret = 0;
4      for (int i=0; i<size; ++i) ret += a[i];
5      return ret;
6  }
7
8  int main() {
9      int a[] = { 1, 2, 3, 4, 5};
10     return sum(a, sizeof(a)/sizeof(a[0]));
11 }
```

Compilemos este ejemplo con máximo nivel de optimización. No es necesario dominar el ensamblador de la arquitectura x86_64 para entender la estructura.

```
$ gcc -S -O3 -c inl.cc
```

El resultado es el siguiente:

Listado 23.3: Resultado de la compilación del ejemplo anterior.

```
1      .file   "inl.cc"
2      .text
3      .p2align 4,,15
4      .globl  _Z3sumPij
5      .type   _Z3sumPij, @function
6  _Z3sumPij:
7  .LFB0:
8      .cfi_startproc
9      xorl    %eax, %eax
10     testl   %esi, %esi
11     pushq   %rbx
12     .cfi_def_cfa_offset 16
13     .cfi_offset 3, -16
```

```
14      je  .L2
15      movq    %rdi, %r8
16      movq    %rdi, %rcx
17      andl    $15, %r8d
18      shrq    $2, %r8
19      negq    %r8
20      andl    $3, %r8d
21      cmpl    %esi, %r8d
22      cmova   %esi, %r8d
23      xorl    %edx, %edx
24      testl   %r8d, %r8d
25      movl    %r8d, %ebx
26      je  .L11
27      .p2align 4,,10
28      .p2align 3
29  .L4:
30      addl    $1, %edx
31      addl    (%rcx), %eax
32      addq    $4, %rcx
33      cmpl    %r8d, %edx
34      jb  .L4
35      cmpl    %r8d, %esi
36      je  .L2
37  .L3:
38      movl    %esi, %r11d
39      subl    %r8d, %r11d
40      movl    %r11d, %r9d
41      shrl    $2, %r9d
42      leal    0(,%r9,4), %r10d
43      testl   %r10d, %r10d
44      je  .L6
45      pxor    %xmm0, %xmm0
46      leaq    (%rdi,%rbx,4), %r8
47      xorl    %ecx, %ecx
48      .p2align 4,,10
49      .p2align 3
50  .L7:
51      addl    $1, %ecx
52      paddd   (%r8), %xmm0
53      addq    $16, %r8
54      cmpl    %r9d, %ecx
55      jb  .L7
56      movdqa  %xmm0, %xmm1
57      addl    %r10d, %edx
58      psrldq  $8, %xmm1
59      paddd   %xmm1, %xmm0
60      movdqa  %xmm0, %xmm1
61      psrldq  $4, %xmm1
62      paddd   %xmm1, %xmm0
63      movd    %xmm0, -4(%rsp)
64      addl    -4(%rsp), %eax
65      cmpl    %r10d, %r11d
66      je  .L2
67  .L6:
68      movslq  %edx, %rcx
69      leaq    (%rdi,%rcx,4), %rcx
70      .p2align 4,,10
71      .p2align 3
72  .L9:
73      addl    $1, %edx
74      addl    (%rcx), %eax
75      addq    $4, %rcx
76      cmpl    %edx, %esi
77      ja  .L9
78  .L2:
79      popq    %rbx
80      .cfi_remember_state
81      .cfi_def_cfa_offset 8
82      ret
83  .L11:
84      .cfi_restore_state
```

```
85      movl    %r8d, %eax
86      jmp .L3
87      .cfi_endproc
88  .LFE0:
89      .size   _Z3sumPij, .-_Z3sumPij
90      .section    .text.startup,"ax",@progbits
91      .p2align 4,,15
92      .globl  main
93      .type   main, @function
94  main:
95  .LFB1:
96      .cfi_startproc
97      movl    $15, %eax
98      ret
99      .cfi_endproc
100 .LFE1:
101     .size   main, .-main
102     .ident  "GCC: (Debian 4.6.3-1) 4.6.3"
103     .section    .note.GNU-stack,"",@progbits
```

El símbolo `_Z3sumPij` corresponde a la función `sum()` aplicando las reglas de *mangling*. Podemos decodificarlo usando `c++filt`.

```
$ echo _Z3sumPij | c++filt
sum(int*, unsigned int)
```

El símbolo codifica la signatura entera de la función. Sin embargo no se utiliza en ninguna parte. Observemos en detalle las instrucciones de la función `main()` eliminando las directivas no necesarias.

Listado 23.4: Código de la función `main()` del ejemplo anterior.

```
1 main:
2         movl    $15, %eax
3         ret
```

El código se limita a retornar el resultado final, un 15. El compilador ha realizado la expansión en línea y sucesivamente ha aplicado propagación de constantes y evaluación de expresiones constantes para simplificarlo a lo mínimo. Y entonces ¿por qué aparece el código de la función `sum()`?

El motivo es simple, la función puede ser necesaria desde otra unidad de compilación. Por ejemplo, supóngase que en otra unidad de compilación aparece el siguiente código.

Listado 23.5: Otra unidad de compilación puede requerir la función `sum()`.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int sum(int* a, unsigned sz);
6
7  struct A {
8      A() {
9          int a[] = { 1, 1, 1, 1 };
10         cout << sum(a, 4) << endl;
11     }
12 };
13
14 A a;
```

¿Significa eso que el código no utilizado ocupa espacio en el ejecutable? Podemos responder a esa pregunta compilando el ejemplo inicial y examinando los símbolos con `nm`:

```
$ g++ -O3 -o inl inl.cc
$ nm --dynamic inl
                 w _Jv_RegisterClasses
                 w __gmon_start__
                 U __libc_start_main
```

No ha quedado ningún símbolo reconocible. El montador ha optimizado el ejecutable para que solo contenga los símbolos utilizados. ¿Y si un plugin necesita la función `sum()`? La respuesta la conocemos, aunque no conocíamos los detalles, basta montar con la opción `-rdynamic`:

```
$ g++ -O3 -rdynamic -o inl inl.cc
$ nm --dynamic inl
00000000004008d8 R _IO_stdin_used
                 w _Jv_RegisterClasses
0000000000400720 T _Z3sumPij
0000000000600c00 A __bss_start
0000000000600bf0 D __data_start
                 w __gmon_start__
00000000004007f0 T __libc_csu_fini
0000000000400800 T __libc_csu_init
                 U __libc_start_main
0000000000600c00 A _edata
0000000000600c10 A _end
00000000004008c8 T _fini
00000000004005f8 T _init
0000000000400638 T _start
0000000000600bf0 W data_start
0000000000400630 T main
```

Si el código está en una biblioteca dinámica el montador no eliminará los símbolos porque no puede determinar si se usarán en el futuro. Sin embargo algunas funciones solo serán necesarias en un archivo concreto. En ese caso pueden declararse como `static`, lo que evita que se exporte el símbolo.

La palabra `static` es seguramente la palabra clave más sobrecargada de C++. Aplicado a las funciones o las variables globales quiere decir que el símbolo no se exporta. Aplicado a un método quiere decir que se trata de un método de clase, no aplicado a una instancia concreta. Aplicado a una variable local quiere decir que se almacena en la zona de datos estáticos.

Listado 23.6: Esta biblioteca solo exporta la función `sum10()`.

```
1  static int sum(int* a, unsigned size)
2  {
3      int ret = 0;
4      for (int i=0; i<size; ++i) ret += a[i];
5      return ret;
6  }
7
8  int sum10(int* a)
9  {
10     return sum(a,10);
11 }
```

La expansión en línea de las funciones no siempre produce un código óptimo. Para ilustrar este punto vamos a utilizar un ejemplo ya conocido de la sección anterior. En dicha sección describíamos un caso de optimización de `git` de Ingo Molnar. Simplificando al máximo el caso se trataba del siguiente fragmento de código:

Listado 23.7: Funciones críticas en la ejecución de `git gc`.

```
1  #include <string.h>
2
3  static inline int hashcmp(const char *sha1, const char *sha2)
4  {
5      return memcmp(sha1, sha2, 20);
6  }
7
8  extern const char null_sha1[20] __attribute__((aligned(8)));
9  static inline int is_null_sha1(const char *sha1)
10 {
11     return !hashcmp(sha1, null_sha1);
12 }
13
14
15 int ejemplo(char* sha1, char* index, unsigned mi)
16 {
17         int cmp, i;
18         for (i=0; i<mi; ++i) {
19                 cmp = hashcmp(index + i * 1024, sha1);
20                 if (cmp == 0) return 0;
21         }
22         return cmp;
23 }
```

Estas funciones, que eran expandidas en línea por el compilador, exhibían un comportamiento anómalo con respecto a los ciclos de estancamiento y a la predicción de saltos. Por lo que Ingo propone la siguiente optimización:

Listado 23.8: Optimización de funciones críticas en la ejecución de `git gc`.

```
1  static inline int hashcmp(const char *sha1, const char *sha2)
2  {
3      int i;
4
5      for (i = 0; i < 20; i++, sha1++, sha2++) {
6          if (*sha1 != *sha2)
7              return *sha1 - *sha2;
8      }
9
10     return 0;
11 }
12
13 extern const char null_sha1[20];
14 static inline int is_null_sha1(const char *sha1)
15 {
16     return !hashcmp(sha1, null_sha1);
17 }
18
19
20 int ejemplo(char* sha1, char* index, unsigned mi)
21 {
22         int cmp, i;
23         for (i=0; i<mi; ++i) {
24                 cmp = hashcmp(index + i * 1024, sha1);
25                 if (cmp == 0) return 0;
26         }
27         return cmp;
28 }
```

Lo interesante de este caso de estudio es que partió de un análisis con el perfilador que determinaba que la función `memcmp()` era subóptima para comparaciones cortas. La función `memcmp()` se expandía automáticamente en línea en forma de un puñado de instrucciones ensamblador. Una de ellas, `repz cmpsb`, era identificada como la culpable del problema. Actualmente ni `gcc-4.6` ni `clang` expanden automáticamente la función `memcmp()`. Por tanto el resultado es bien distinto. Empleando `perf stat -r 100 -e cycles:u` se obtienen los resultados que muestra la tabla 23.3.

Compilador	Ciclos	Ciclos Opt.	Mejora
`gcc-4.6`	192458	190022	1,3 %
`clang-3.0`	197163	198232	-0,5 %
`llvm-gcc-4.6`	189164	191826	-1,4 %

Cuadro 23.3: Resultados de la optimización de Ingo Molnar con compiladores actuales (100 repeticiones).

El mejor resultado lo obtiene `llvm-gcc` con el caso sin optimizar. El caso de `clang` genera resultados absolutamente comparables, dentro de los márgenes de error de `perf`. En cualquiera de los casos el resultado es mucho menos significativo que los resultados que obtuvo Ingo Molnar. Una optimización muy efectiva en un contexto puede no ser tan efectiva en otro, y el contexto es siempre cambiante (nuevas versiones de los compiladores, nuevas arquitecturas, etc.).

23.2.3. Eliminación de copias

En la mayor parte del estándar de C++ se suele indicar que el compilador tiene libertad para optimizar siempre que el resultado se comporte *como si* esas optimizaciones no hubieran tenido lugar. Sin embargo el estándar permite además un rango de optimizaciones muy concreto pero con gran impacto en prestaciones, que pueden cambiar el comportamiento de un programa. En [49], sección 12.8, § 32 introduce la noción de *copy elision*. Lo que sigue es una traducción literal del estándar.

> *Cuando se cumplen determinados criterios una implementación puede omitir la llamada al constructor de copia o movimiento de un objeto, incluso cuando el constructor y/o destructor de dicho objeto tienen efectos de lado. En estos casos la implementación simplemente trata la fuente y el destino de la operación de copia o movimiento omitida como dos formas diferentes de referirse al mismo objeto, y la destrucción de dicho objeto ocurre cuando ambos objetos hubieran sido destruidos sin la optimización. Esta elisión de las operaciones de copia o movimiento, denominada* elisión de copia*, se permite en las siguientes circunstancias (que pueden ser combinadas para eliminar copias múltiples):*
>
> - *En una sentencia* return *de una función cuyo tipo de retorno sea una clase, cuando la expresión es el nombre de un objeto automático no volátil (que no sea un parámetro de función o un parámetro de una cláusula* catch*) con el mismo tipo de retorno de la función (que no puede ser* const *ni* volatile*), la operación de copia o movimiento puede ser omitida mediante la construcción directa del objeto automático en el propio valor de retorno de la función.*
> - *En una expresión* throw*, cuando el operando es el nombre de un objeto automático no volátil (que no sea un parámetro de función o un parámetro de una cláusula* catch*) cuyo ámbito de declaración no se extienda más allá del final del bloque* try *más interior que contenga a*

dicha expresión (si es que existe), la operación de copia o movimiento desde el operando hasta el objeto excepción puede ser omitida mediante la construcción del objeto automático directamente sobre el objeto excepción.

- *Cuando un objeto temporal de clase que no ha sido ligado a una referencia sería copiado o movido a un objeto con la misma calificación de* const/volatile*, la operación de copia o movimiento puede ser omitida construyendo el temporal directamente sobre el destino de la copia o movimiento.*
- *Cuando la declaración de excepción en una clausula* catch *declara un objeto del mismo tipo (salvo por modificadores* const *o* volatile*) como el objeto excepción, la operación de copia o movimiento puede ser omitida tratando la declaración de excepción como un alias del objeto excepción siempre que el significado del programa no sea cambiado salvo por la ejecución de constructores y destructores del objeto de la declaración de excepción.*

```
1     class Thing {
2       public:
3       Thing();
4       ~Thing();
5       Thing(const Thing&);
6     };
7
8     Thing f() {
9       Thing t;
10      return t;
11    }
12
13    Thing t2 = f();
```

Aquí los criterios de elisión pueden combinarse para eliminar dos llamadas al constructor de copia de `Thing`*: la copia del objeto automático local* `t` *en el objeto temporal para el valor de retorno de la función* `f()` *y la copia de ese objeto temporal al objeto* `t2`*. Por tanto la construcción del objeto local* `t` *puede verse como la inicialización directa del objeto* `t2`*, y la destrucción de dicho objeto tendrá lugar al terminar el programa. Añadir un constructor de movimiento a* `Thing` *tiene el mismo efecto, en cuyo caso es el constructor de movimiento del objeto temporal a* `t2` *el que se elide.*

Copy elision es un concepto que incluye dos optimizaciones frecuentes en compiladores de C++: RVO (tercera circunstancia contemplada en el estándar) y NRVO (Named Return Value Optimization) (primera circunstancia contemplada en el estándar).

23.2.4. Volatile

Las optimizaciones del compilador pueden interferir con el funcionamiento del programa, especialmente cuando necesitamos comunicarnos con periféricos. Así por ejemplo, el compilador es libre de reordenar y optimizar las operaciones mientras mantenga una equivalencia funcional. Así, por ejemplo este caso se encuentra no pocas veces en código de videojuegos caseros para consolas.

```
1 void reset(unsigned& reg)
2 {
3   reg = 1;
4   for(int i=0; i<1000000; ++i);
5   reg = 0;
6 }
```

El programador piensa que el bucle implementa un retardo y por tanto la función permite generar un pulso en el bit menos significativo. Compilando el ejemplo con máximo nivel de optimización obtenemos lo siguiente:

```
1 _Z5resetRj:
2         movl    $0, (%rdi)
3         ret
```

El compilador ha eliminado todo hasta el punto de que ni siquiera escribe el pulso. Una forma sencilla de corregir este comportamiento es declarar el contador `i` y el registro `reg` como `volatile`. Esto indica al compilador que no debe hacer optimizaciones con respecto a esas variables. Otra forma sería sustituir el bucle de espera por una llamada a función (por ejemplo `usleep(10)`).

23.3. Conclusiones

La optimización de programas es una tarea sistemática, pero a la vez creativa. Toda optimización parte de un análisis cuantitativo previo, normalmente mediante el uso de perfiladores. Existe un buen repertorio de herramientas que nos permite caracterizar las mejores oportunidades, pero no todo lo que consume tiempo está en nuestra mano cambiarlo. Las mejores oportunidades provienen de la re-estructuración de los algoritmos o de las estructuras de datos.

Por otro lado el programador de videojuegos deberá optimizar para una plataforma o conjunto de plataformas que se identifican como objetivo. Algunas de las optimizaciones serán específicas para estas plataformas y deberán re-evaluarse cuando el entorno cambie.

Capítulo 24

Validación y Pruebas

David Villa Alises

La programación, igual que cualquier otra disciplina técnica, debería ofrecer garantías sobre los resultados. La formalización de algoritmos ofrece garantías indiscutibles y se utiliza con éxito en el ámbito de los algoritmos numéricos y lógicos. Sin embargo, el desarrollo de software en muchos ámbitos está fuertemente ligado a requisitos que provienen directamente de necesidades de un cliente. En la mayoría de los casos, esas necesidades no se pueden formalizar dado que el cliente expresa habitualmente requisitos ambiguos o incluso contradictorios. El desarrollo de software no puede ser ajeno a esa realidad y debe integrar al cliente de forma que pueda ayudar a validar, refinar o rectificar la funcionalidad del sistema durante todo el proceso.

Un programador responsable comprueba que su software satisface los requisitos del cliente, comprueba los casos típicos y se asegura que los errores detectados (ya sea durante el desarrollo o en producción) se resuelven y no vuelven a aparecer. Es imposible escribir software perfecto (a día de hoy) pero un programador realmente profesional escribe código limpio, legible, fácil de modificar y adaptar a nuevas necesidades. En este capítulo veremos algunas técnicas que pueden ayudar a escribir código más limpio y robusto.

24.1. Programación defensiva

La expresión «programación defensiva» se refiere a las técnicas que ayudan al programador a evitar, localizar y depurar fallos, especialmente aquellos que se producen en tiempo de ejecución. En muchas situaciones, especialmente con lenguajes como C y C++, el programa puede realizar una operación ilegal que puede terminar con la ejecución del proceso por parte del sistema operativo. El caso más conocido en este

sentido se produce cuando se dereferencia un puntero que apunta a memoria fuera de los límites reservados para ese proceso: el resultado es el fatídico mensaje *segmentation fault* (abreviado como SEGFAULT). Cuando esa situación no ocurre en todos los casos sino que aparece esporádicamente, encontrar la causa del problema puede ser realmente complicado y puede llevar mucho tiempo. Para este tipo de problemas la depuración *postmortem* es una gran ayuda, pero antes de llegar a la autopsia, hay algunas medidas preventivas que podemos tomar: el control de invariantes.

En programación, una invariante es un predicado que asumimos como cierto antes, durante y después de la ejecución de un bloque de código (típicamente una función o método). Definir invariantes en nuestras funciones puede ahorrar mucho tiempo de depuración porque tenemos garantías de que el problema está limitado al uso correcto de la función que corresponda.

Muy ligado al concepto de invariante existe una metodología denominada «diseño por contrato». Se trata de un método para definir la lógica de una función, objeto u otro componente de modo que su interfaz no depende solo de los tipos de sus parámetros y valor de retorno. Se añaden además predicados que se evalúan antes (pre-condiciones) y después (post-condiciones) de la ejecución del bloque de código. Así, la interfaz de la función es mucho más rica, el valor del parámetro además de ser del tipo especificado debe tener un valor que cumpla con restricciones inherentes al problema.

Listado 24.1: Una función que define una *invariante* sobre su parámetro

```
1 double sqrt(double x) {
2   assert(x >= 0);
3   [...]
4 }
```

Normalmente el programador añade comprobaciones que validan los datos de entrada procedentes de la interfaz con el usuario. Se trata principalmente de convertir y verificar que los valores obtenidos se encuentran dentro de los rangos o tengan valores según lo esperado. Si no es así, se informa mediante la interfaz de usuario que corresponda. Sin embargo, cuando se escribe una función que va a ser invocada desde otra parte, no se realiza una validación previa de los datos de entrada ni tampoco de los producidos por la función. En condiciones normales podemos asumir que la función va a ser invocada con los valores correctos, pero ocurre que un error en la lógica del programa o un simple error-por-uno puede implicar que utilicemos incorrectamente nuestras propias funciones, provocando errores difíciles de localizar.

La herramienta más simple, a la vez que potente, para definir invariantes, pre-condiciones o post-condiciones es la función `assert()`[1], que forma parte de la librería estándar de prácticamente todos los lenguajes modernos. `assert()` sirve, tal como indica su nombre, para definir aserciones, que en el caso de C++ será toda expresión que pueda ser evaluada como cierta. El siguiente listado es un ejemplo mínimo de usa aserción. Se muestra también el resultado de ejecutar el programa cuando la aserción falla:

Listado 24.2: `assert-argc.cc`: Un ejemplo sencillo de `assert()`

```
1 #include <cassert>
2
3 int main(int argc, char *argv[]) {
4   assert(argc == 2);
5   return 0;
6 }
```

«Error por uno»

Se denomina así a los errores (*bugs*) debidos a comprobaciones incorrectas ('>' por '>=', '<' por '<=' o viceversa), en la indexación de vectores en torno a su tamaño, iteraciones de bucles, etc. Estos casos deben ser objeto de testing concienzudo.

[1] En C++. la función `assert()` se encuentra en el fichero de cabecera `<cassert>`.

```
$ ./assert-argc hello
$ ./assert-argc
assert-argc: assert-argc.cc:4: int main(int, char**): Assertion `argc
    == 2' failed.
Abortado
```

Veamos algunos usos habituales de `assert()`

- Validar los parámetros de una función (pre-condiciones). Por ejemplo, comprobar que una función recibe un puntero no nulo:

```
1 void Inventory::add(Weapon* weapon) {
2   assert(weapon);
3   [...]
4 }
```

- Comprobar que el estado de un objeto es consistente con la operación que se está ejecutando, ya sea como pre-condición o como post-condición.
- Comprobar que un algoritmo produce resultados consistentes. Este tipo de post-condiciones se llaman a menudo *sanity checks*.
- Detectar condiciones de error irrecuperables.

```
1 void Server::bind(int port) {
2   assert(port > 1024);
3   assert(not port_in_use(port));
4   [...]
5 }
```

24.1.1. Sobrecarga

Las aserciones facilitan la depuración del programa porque ayudan a localizar el punto exacto donde se desencadena la inconsistencia. Por eso deberían incluirse desde el comienzo de la implementación. Sin embargo, cuando el programa es razonablemente estable, las aserciones siempre se cumplen (o así debería ser). En una versión de producción las aserciones ya no son útiles[2] y suponen una sobrecarga que puede afectar a la eficiencia del programa.

Obviamente, eliminar «a mano» todas las aserciones no parece muy cómodo. La mayoría de los lenguajes incorporan algún mecanismo para desactivarlas durante la compilación. En C/C++ se utiliza el preprocesador. Si la constante simbólica `NDEBUG` está definida la implementación de `assert()` (que en realidad es una macro de preprocesador) se substituye por una sentencia vacía de modo que el programa que se compila realmente no tiene absolutamente nada referente a las aserciones.

En los casos en los que necesitamos hacer aserciones más complejas, que requieran variables auxiliares, podemos aprovechar la constante `NDEBUG` para eliminar también ese código adicional cuando no se necesite:

```
1 [...]
2 #ifndef NDEBUG
3 vector<int> values = get_values();
4 assert(values.size());
5 #endif
```

[2] Por contra, algunos autores como Tony Hoare, defienden que en la versión de producción es dónde más necesarias son las aserciones.

Aunque esta contante se puede definir simplemente con `#define NDEBUG`, lo más cómodo y aconsejable es utilizar el soporte que los compiladores suelen ofrecer para definir contantes en línea de comandos. En el caso de `g++` se hace así:

```
$ g++ -DNDEBUG main.cc
```

Definir la constante en el código, aparte de ser incómodo cuando se necesita activar/desactivar con frecuencia, puede ser confuso porque podría haber ficheros que se preprocesan antes de que la constante sea definida.

24.2. Desarrollo ágil

El desarrollo ágil de software trata de reducir al mínimo la burocracia típica de las metodologías de desarrollo tradicionales. Se basa en la idea de que «el software que funciona es la principal medida de progreso». El desarrollo ágil recoge la herencia de varías corrientes de finales de los años 90 como Scrum o la programación extrema y todas esas ideas se plasmaron en el llamado *manifiesto ágil*:

> Estamos descubriendo formas mejores de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de este trabajo hemos aprendido a valorar:
>
> - *Individuos e interacciones* sobre procesos y herramientas.
> - *Software funcionando* sobre documentación extensiva.
> - *Colaboración con el cliente* sobre negociación contractual.
> - *Respuesta ante el cambio* sobre seguir un plan.
>
> Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda.

Figura 24.1: Kent Beck, uno de los principales creadores de *eXtreme programing*, TDD y los métodos ágiles.

Las técnicas de desarrollo ágil pretenden entregar valor al cliente pronto y a menudo, es decir, priorizar e implementar las necesidades expresadas por el cliente para ofrecerle un producto que le pueda resultar útil desde el comienzo. También favorecen la adopción de cambios importantes en los requisitos, incluso en las últimas fases del desarrollo.

24.3. TDD

Una de las técnicas de desarrollo ágil más efectiva es el Desarrollo Dirigido por Pruebas o TDD (Test Driven Development). La idea básica consiste en empezar el proceso escribiendo pruebas que representen directamente requisitos del cliente. Algunos autores creen que el término «ejemplo» describe mejor el concepto que «prueba». Una prueba es un pequeño bloque de código que se ejecuta sin ningún tipo de interacción con el usuario (ni entrada ni salida) y que determina de forma inequívoca (la prueba pasa o falla) si el requisito correspondiente se está cumpliendo.

En el desarrollo de software tradicional las pruebas se realizan una vez terminado el desarrollo asumiendo que desarrollo y pruebas son fases estancas. Incluso en otros modelos como el iterativo, en espiral o el prototipado evolutivo las pruebas se realizan después de la etapa de diseño y desarrollo, y en muchas ocasiones por un equipo de programadores distinto al que ha escrito el código.

24.3.1. Las pruebas primero

Con TDD la prueba es el primer paso que desencadena todo el proceso de desarrollo. En este sentido, las pruebas no son una mera herramienta de *testing*. Las pruebas se utilizan como un medio para capturar y definir con detalle los requisitos del usuario, pero también como ayuda para obtener un diseño consistente evitando añadir complejidad innecesaria. Hacer un desarrollo dirigido por pruebas acota el trabajo a realizar: si todas las pruebas pasan, el programa está terminado, algo que puede no resultar trivial con otros modelos de desarrollo.

Este proceso resulta muy útil para evitar malgastar tiempo y esfuerzo añadiendo funcionalidad que en realidad no se ha solicitado. Este concepto se conoce como YAGNI (You Ain't Gonna Need It) y aunque a primera vista pueda parecer una cuestión trivial, si se analiza detenidamente, puede suponer un gran impacto en cualquier proyecto. Es frecuente que los programadores entusiastas y motivados por la tarea acaben generando un diseño complejo plasmado en una gran cantidad de código difícil de mantener, mejorar y reparar.

24.3.2. rojo, verde, refactorizar

Cada uno de los requisitos identificados debe ser analizado hasta obtener una serie de escenarios que puedan ser probados de forma independiente. Cada uno de esos escenarios se convertirá en una prueba. Para cada uno de ellos:

- Escribe la prueba haciendo uso de las interfaces del sistema (¡es probable que aún no existan!) y ejecútala. La prueba debería fallar y debes comprobar que es así (**rojo**).
- A continuación escribe el código de producción mínimo necesario para que la prueba pase (**verde**). Ese código «mínimo» debe ser solo el imprescindible, lo más simple posible, hasta el extremo de escribir métodos que simplemente retornan el valor que la prueba espera[3]. Eso ayuda a validar la interfaz y confirma que la prueba está bien especificada. Pruebas posteriores probablemente obligarán a modificar el código de producción para que pueda considerar todas las posibles situaciones. A esto se le llama «triangulación» y es la base de TDD: Las pruebas dirigen el diseño.
- Por último **refactoriza** si es necesario. Es decir, revisa el código de producción y elimina cualquier duplicidad. También es el momento adecuado para renombrar tipos, métodos o variables si ahora se tiene más claro cuál es su objetivo real. Por encima de cualquier otra consideración el código debe expresar claramente la *intención del programador*. Es importante refactorizar tanto el código de producción como las propias pruebas.

Este sencillo método de trabajo (el algoritmo TDD) favorece que los programadores se concentren en lo que realmente importa: satisfacer los requisitos del usuario. También ayuda al personal con poca experiencia en el proyecto a decidir *cuál es el próximo paso* en lugar de divagar o tratando de «mejorar» el programa añadiendo funcionalidades no solicitadas.

[3] Kent Beck se refiere a esto con la expresión "Fake until make it".

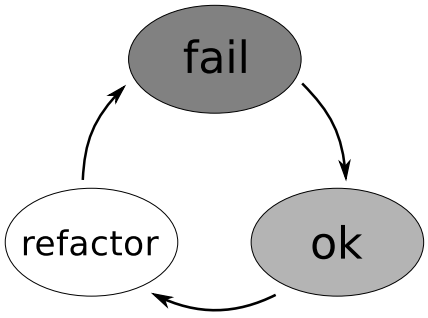


Figura 24.2: Algoritmo TDD

24.4. Tipos de pruebas

Hay muchas formas de clasificar las pruebas, y todo lo referente al *testing* tradicional es aplicable aquí, aunque quizá de un modo diferente: pruebas de caja negra y blanca, pruebas de aceptación, integración, sistema, unitarias y largo etcétera. En el contexto de las metodologías ágiles podemos concretar los siguientes tipos de pruebas [13]:

De aceptación Idealmente debería estar especificado por el cliente o al menos por un analista con la ayuda del cliente. Se expresa en términos del dominio de la aplicación, sin detalles de implementación. Esto incluye los test no funcionales, es decir, aquellos que expresan requisitos no relacionados con los resultados obtenidos sino sobre cuestiones como tiempo de ejecución, consumo de energía, etc.

De sistema Es un test que utiliza el sistema completo, desde el interfaz de usuario hasta la base de datos, y lo hace del mismo modo que lo harían los usuarios reales. Son pruebas muy frágiles, es decir, pequeños cambios sin relación aparente pueden hacer fallar la prueba aunque funcionalmente el sistema sea correcto.

Unitarios Se utilizan para probar un único componente del sistema: un método o función, y para unas condiciones concretas (un escenario). La validación se puede hacer bien comprobando el estado final conocido el estado inicial o por la interacción entre el componente que se está probando y sus colaboradores.

Desde el punto de vista ágil hay una pauta clara: las pruebas se escriben para ejecutarse, y debería ocurrir tan a menudo como sea posible. Lo ideal sería ejecutar todas las pruebas después de cada cambio en cualquier parte de la aplicación. Obviamente eso resulta prohibitivo incluso para aplicaciones pequeñas. Hay que llegar a una solución de compromiso. Por este motivo, las pruebas unitarias son las más importantes. Si están bien escritas, las pruebas unitarias se deberían poder ejecutar en muy pocos segundos. Eso permite que, con los frameworks y herramientas adecuadas se pueda lanzar la batería de pruebas unitarias completa mientras se está editando el código (sea la prueba o el código de producción).

Para que una prueba se considere unitaria no basta con que esté escrita en un framework xUnit, debe cumplir los principios FIRST [60], que es un acrónimo para:

Fast Las pruebas unitarias deberían ser muy rápidas. Como se ha dicho, todas las pruebas unitarias de la aplicación (o al menos del módulo) deberían ejecutarse en menos de 2–3 segundos.

Independent Cada prueba debe poder ejecutarse por separado o en conjunto, y en cualquier orden, sin que eso afecte al resultado.

Repeatable La prueba debería poder ejecutarse múltiples veces dando siempre el mismo resultado. Por este motivo no es buena idea incorporar aleatoriedad a los tests. También implica que la prueba debe poder funcionar del mismo modo en entornos distintos.

Self-validating La prueba debe ofrecer un resultado concreto: pasa o falla, sin que el programador tenga que leer o interpretar un valor en pantalla o en un fichero.

Timely El test unitario debería escribirse justo cuando se necesite, es decir, justo antes de escribir el código de producción relacionado, ni antes ni después.

En cuanto al resto de las pruebas: sistema, integración y aceptación; deberían ejecutarse al menos una vez al día. Existe toda una disciplina, llamada «integración continua» que trata sobre la compilación, integración y prueba de todo el sistema de forma totalmente automática, incluyendo la instalación de dependencias e incluso el empaquetado y despliegue. Esta operación puede hacerse cada vez que un programador añade nuevo código al repositorio o bien una vez al día si la aplicación es muy grande. El objetivo es disponer de información precisa y actualizada sobre el estado de la aplicación en su conjunto y sobre los requisitos que está cumpliendo.

24.5. Pruebas unitarias con google-tests

En esta sección veremos un ejemplo de TDD intencionadamente simple para crear la función `factorial()`[4]. Para ello vamos a utilizar el framework de pruebas google-tests (gtest). El primer test prueba que el resultado de `factorial(0)` es 1:

Listado 24.3: `factorial-test.cc`: Pruebas para `factorial()`

```
1 #include "gtest/gtest.h"
2 #include "factorial.h"
3
4 TEST(FactorialTest, Zero) {
5   EXPECT_EQ(1, factorial(0));
6 }
```

Se incluye el archivo de cabecera de `gtest` (línea 1) donde están definidas las macros que vamos a utilizar para definir las pruebas. La línea 4 define una prueba llamada `Zero` mediante la macro `TEST` para el casa de prueba (*TestCase*) `FactorialTest`. La línea 5 especifica una expectativa: el resultado de invocar `factorial(0)` debe ser igual a 1.

Además del fichero con la prueba debemos escribir el código de producción, su fichero de cabecera y un `Makefile`. Al escribir la expectativa ya hemos decidido el nombre de la función y la cantidad de parámetros (aunque no es tipo). Veamos estos ficheros:

[4] Inspirado en el primer ejemplo del tutorial de GTests.

Listado 24.4: Escribiendo `factorial()` con TDD: `Makefile`

```
CC=$(CXX)
CXXFLAGS=-Igtest/include -Igtest -pthread
LDFLAGS=-pthread -lpthread

GTEST_TARBALL=gtest-1.7.0.zip

all: factorial-test

factorial-test: factorial-test.o factorial1.o gtest_main.a

gtest_main.a: gtest-all.o gtest_main.o
    ar rv $@ $^

gtest%.o: gtest/src/gtest%.cc
    g++ $(CXXFLAGS) -c $<

gtest: $(GTEST_TARBALL)
    unp $<
    ln -s $(GTEST_TARBALL:.zip=) gtest

clean:
    $(RM) factorial-test *.o *~
    $(RM) *.a
```

Listado 24.5: Escribiendo `factorial()` con TDD: `factorial.h`

```
int factorial(int n);
```

Y el código de producción mínimo para pasar la prueba:

Listado 24.6: Escribiendo `factorial()` con TDD: `factorial.cc` (1)

```
#include "factorial.h"

int factorial(int n) {
  return 1;
}
```

Compilamos y ejecutamos el binario obtenido:

```
$ make
g++ -c -o factorial.o factorial.cc
g++ factorial-test.o factorial.o -lpthread -lgtest -lgtest_main -o
    factorial-test
$ ./factorial-test
Running main() from gtest_main.cc
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from FactorialTest
[ RUN      ] FactorialTest.Zero
[       OK ] FactorialTest.Zero (0 ms)
[----------] 1 test from FactorialTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
```

Esta primera prueba no la hemos visto fallar porque sin el fichero de producción ni siquiera podríamos haberla compilado.

Añadamos ahora un segundo caso de prueba al fichero:

Listado 24.7: `factorial-test.cc`: Pruebas para `factorial()`

```
1 TEST(FactorialTest, Positive) {
2   EXPECT_EQ(1, factorial(1));
3   EXPECT_EQ(2, factorial(2));
4 }
```

Como es lógico, la expectativa de la línea 2 también pasa ya que el resultado es el mismo que para la entrada 0. Veamos el resultado:

```
1  $ ./factorial-test
2  Running main() from gtest_main.cc
3  [==========] Running 2 tests from 1 test case.
4  [----------] Global test environment set-up.
5  [----------] 2 tests from FactorialTest
6  [ RUN      ] FactorialTest.Zero
7  [       OK ] FactorialTest.Zero (0 ms)
8  [ RUN      ] FactorialTest.Positive
9  factorial-test.cc:10: Failure
10 Value of: factorial(2)
11   Actual: 1
12 Expected: 2
13 [  FAILED  ] FactorialTest.Positive (0 ms)
14 [----------] 2 tests from FactorialTest (0 ms total)
15
16 [----------] Global test environment tear-down
17 [==========] 2 tests from 1 test case ran. (0 ms total)
18 [  PASSED  ] 1 test.
19 [  FAILED  ] 1 test, listed below:
20 [  FAILED  ] FactorialTest.Positive
21
22  1 FAILED TEST
```

El resultado nos indica la prueba que ha fallado (`línea 8`), el valor obtenido por la llamada (línea 11) y el esperado (línea 12).

A continuación se debe modificar la función `factorial()` para que cumpla la nueva expectativa. Después escribir nuevas pruebas que nos permitan comprobar que la función cumple con su cometido para unos cuantos casos representativos.

24.6. Dobles de prueba

TDD, y el *agilismo* en general, está muy relacionada con la orientación a objetos, y en muchos sentidos se asume que estamos haciendo un diseño orientado a objetos prácticamente en todos los casos.

La mayoría de los lenguajes de programación que soportan la orientación a objetos tienen herramientas para encapsulación y ocultación. La ocultación (el hecho de no exponer los detalles de implementación de la clase) resulta crucial en un diseño orientado a objetos porque proporciona «sustituibilidad» (LSP (Liskov Substitution Principle)). Pero la ocultación dificulta la definición de pruebas en base a aserciones sobre de estado porque el estado del objeto está definido por el valor de sus atributos. Si tuviéramos acceso a todos los atributos del objeto (sea con *getters* o no) sería una pista de un mal diseño.

Debido a ello, suele ser más factible definir la prueba haciendo aserciones sobre la interacción que el objeto que se está probando (el SUT (Subject Under Test)[5]) realiza con sus colaboradores. El problema de usar un colaborador real es que éste tendrá a su vez otros colaboradores de modo que es probable que para probar un único método necesitemos montar gran parte de la aplicación. Como lo que queremos es instanciar lo mínimo posible del sistema real podemos recurrir a los dobles[6] de prueba.

Un doble de prueba es un objeto capaz de simular la interfaz que un determinado colaborador ofrece al SUT, pero que realmente no implementa nada de lógica. El doble (dependiendo de su tipo) tiene utilidades para comprobar qué métodos y parámetros usó el SUT cuando invocó al doble.

SOLID

SOLID (SRP, OCP, LSP, ISP, DIP)[7] es una serie de 5 principios esenciales para conseguir diseños orientados a objetos de calidad. Estos son:

SRP — Single Responsibility
OCP — Open Closed
LSP — Liskov Substitution
DIP — Dependency Inversion
ISP — Interface Segregation

Una regla básica: Nunca se deben crear dobles para clases implementadas por terceros, sólo para clases de la aplicación.

Un requisito importante para poder realizar pruebas con dobles es que las clases de nuestra aplicación permitan «inyección de dependencias». Consiste en pasar (inyectar) las instancias de los colaboradores (dependencias) que el objeto necesitará en el momento de su creación. Pero no estamos hablando de un requisito impuesto por las pruebas, se trata de otro de los principios SOLID, en concreto DIP (Dependency Inversion Principle).

Aunque hay cierta confusión con la terminología, hay bastante consenso en distinguir al menos entre los siguientes tipos de dobles:

Fake Es una versión rudimentaria del objeto de producción. Funcionalmente equivalente, pero tomando atajos que no serían admisibles en el código final. Por ejemplo, una base de datos cuya persistencia es un diccionario en memoria.

Stub Devuelve valores predefinidos para los métodos que el SUT va a invocar. Se trata de un colaborador que «le dice al SUT lo que necesita oir» pero nada más.

Mock El mock se programa con una serie de expectativas (invocaciones a sus métodos) que debería cumplirse durante la ejecución de la prueba. Si alguna de esas llamadas no se produce, u ocurre en una forma diferente a lo esperado, la prueba fallará.

Spy El spy es un objeto que registra todas las invocaciones que se hacen sobre él. Después de utilizado, se pueden hacer aserciones para comprobar que ciertas llamadas a sus métodos ocurrieron. A diferencia del mock, puede haber recibido otras invocaciones además de las que se compruebas y el comportamiento sigue siendo válido.

24.7. Dobles de prueba con google-mock

En esta sección veremos un ejemplo muy simple de uso con google-mock, el framework de dobles C++ que complementa a google-test. Vamos a implementar el método `notify()` de la clase `Observable` (también llamada `Subject`) del patrón *observador*.

[5] SUT: Subject Under Test

[6] Son «dobles» en el mismo sentido que los actores que ruedan las escenas arriesgadas en el cine.

Los primeros ficheros que se muestran son el fichero de cabecera `observable.h`:

Listado 24.8: Patrón observador con TDD: `observable.h`

```
1  #ifndef _OBSERVABLE_H_
2  #define _OBSERVABLE_H_
3
4  #include <vector>
5  #include "observer.h"
6
7  class Observable {
8    std::vector<Observer*> observers;
9  public:
10   void attach(Observer* observer);
11   void detach(Observer* observer);
12   void notify(void);
13 };
14
15 #endif
```

Y el fichero de implementación `observable.cc`:

Listado 24.9: Patrón observador con TDD: `observable.cc`

```
1  #include <algorithm>
2  #include <functional>
3
4  #include "observable.h"
5  #include "observer.h"
6
7  void
8  Observable::attach(Observer* observer) {
9    observers.push_back(observer);
10 }
11
12 void
13 Observable::detach(Observer* observer) {
14   observers.erase(find(observers.begin(), observers.end(),
15                        observer));
16 }
17
18 void
19 Observable::notify(void) {
20   observers[0]->update();
21 }
```

Para escribir un test que pruebe el método `notify()` necesitamos un *mock* para su colaborador (el *observador*). El siguiente listado muestra la interfaz que deben implementar los observadores:

Listado 24.10: Patrón observador con TDD: `observer.h`

```
1  #ifndef _OBSERVER_H_
2  #define _OBSERVER_H_
3
4  class Observer {
5  public:
6    virtual void update(void) = 0;
7    virtual ~Observer() {}
8  };
9
10 #endif
```

Con ayuda de google-mock escribimos el *mock* para este colaborador:

Listado 24.11: Patrón observador con TDD: `mock-observer.h`

```
1  #ifndef MOCK_OBSERVER_H
2  #define MOCK_OBSERVER_H
3
4  #include <gmock/gmock.h>
5  #include "observer.h"
6
7  class MockObserver : public Observer {
8  public:
9    MOCK_METHOD0(update, void());
10 };
11
12 #endif
```

Lo interesante aquí es la definición del método *mockeado* en la línea 9. La macro `MOCK_METHOD0` indica que es para un método sin argumentos llamado `update()` que devuelve `void`. Aunque podemos escribir este fichero *a mano* sin demasiados problemas, existe una herramienta llamada `gmock_gen` que los genera automáticamente a partir de los ficheros de declaración de las clases.

Es hora de escribir la prueba. Vamos a comprobar que si tenemos *observable* con un *observador* registrado e invocamos su método `notify()` el método `update()` del observador se ejecuta una vez (y solo una).

Listado 24.12: Patrón observador con TDD: `observable-tests.cc`

```
1  #include <gmock/gmock.h>
2  #include <gtest/gtest.h>
3
4  #include "observable.h"
5  #include "mock-observer.h"
6
7  TEST(ObserverTest, UpdateObserver) {
8    MockObserver observer;
9    EXPECT_CALL(observer, update()).Times(1);
10
11   Observable observable;
12   observable.attach(&observer);
13
14   observable.notify();
15 }
```

En la prueba creamos el doble para el observer (línea 8) y creamos la expectativa (línea 9). Después creamos el observable (línea 11) y registramos el observador (línea 12). Por último invocamos el método `notify()` (línea 14).

También necesitamos un `Makefile` para compilar y ejecutar la prueba:

Listado 24.13: Patrón observador con TDD: `Makefile`

```
1  GMOCK_DIR = gmock
2  GTEST_DIR = gmock/gtest
3
4  CC        = g++
5  CXXFLAGS  = -I $(GTEST_DIR)/include -I $(GTEST_DIR) -I $(GMOCK_DIR)
       /include -I $(GMOCK_DIR) -pthread
6  LDLIBS    = -pthread -lpthread
7
8  TARGET    = observable-tests
9
10 vpath %.cc $(GMOCK_SRC)/src
11
12 $(TARGET): observable-tests.o observable.o libgmock.a
13
```

```
14 gmock: gmock-1.7.0.zip
15     unp $<
16     ln -s gmock-1.7.0 gmock
17
18 gmock-1.7.0.zip:
19     wget https://googlemock.googlecode.com/files/gmock-1.7.0.zip
20
21 libgmock.a: gtest-all.o gmock-all.o gmock_main.o
22     ar rv $@ $^
23
24 gtest-all.o: $(GTEST_DIR)/src/gtest-all.cc
25     g++ $(CXXFLAGS) -c $<
26
27 gmock%.o: $(GMOCK_DIR)/src/gmock%.cc
28     g++ $(CXXFLAGS) -c $<
29
30 test: $(TARGET)
31     ./$<
32
33 clean:
34     $(RM) $(TARGET) *.o *~ *.a
```

Ejecutemos el `Makefile`:

```
1  $ make test
2  g++ -I /usr/src/gmock   -c -o observable-tests.o observable-tests.cc
3  g++ -I /usr/src/gmock   -c -o observable.o observable.cc
4  g++ -I /usr/src/gmock   -c -o gmock_main.o /usr/src/gmock/src/
       gmock_main.cc
5  g++ -I /usr/src/gmock   -c -o gmock-all.o /usr/src/gmock/src/gmock-all
       .cc
6  g++  observable-tests.o observable.o gmock_main.o gmock-all.o  -
       lpthread -lgtest -o observable-tests
7  $ ./observable-tests
8  Running main() from gmock_main.cc
9  [==========] Running 1 test from 1 test case.
10 [----------] Global test environment set-up.
11 [----------] 1 test from ObserverTest
12 [ RUN      ] ObserverTest.UpdateObserver
13 observable-tests.cc:11: Failure
14 Actual function call count doesn't match EXPECT_CALL(observer, update
       ())...
15          Expected: to be called once
16            Actual: never called - unsatisfied and active
17 [  FAILED  ] ObserverTest.UpdateObserver (1 ms)
18 [----------] 1 test from ObserverTest (1 ms total)
19
20 [----------] Global test environment tear-down
21 [==========] 1 test from 1 test case ran. (1 ms total)
22 [  PASSED  ] 0 tests.
23 [  FAILED  ] 1 test, listed below:
24 [  FAILED  ] ObserverTest.UpdateObserver
25
26  1 FAILED TEST
```

Después de la compilación (líneas 2-6) se ejecuta el binario correspondiente al test (línea 7). El test falla porque se esperaba una llamada a `update()` (línea 15) y no se produjo ninguna (línea 16) de modo que la expectativa no se ha cumplido. Es lógico porque el cuerpo del método `notify()` está vacío. Siguiendo la filosofía TDD escribir el código mínimo para que la prueba pase:

Listado 24.14: Código mínimo para satisfacer la expectativa

```
1 void
2 Observable::notify(void) {
3   observers[0]->update();
4 }
```

Volvemos a ejecutar la prueba:

```
$ make test
g++ -I /usr/src/gmock   -c -o observable.o observable.cc
g++   observable-tests.o observable.o gmock_main.o gmock-all.o  -
    lpthread -lgtest -o observable-tests
$ ./observable-tests
Running main() from gmock_main.cc
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from ObserverTest
[ RUN      ] ObserverTest.UpdateObserver
[       OK ] ObserverTest.UpdateObserver (0 ms)
[----------] 1 test from ObserverTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
```

La prueba pasa. Hora de escribir otra prueba. Comprobemos que `update()` no se invoca si nadie invoca `notify()`:

Listado 24.15: Prueba negativa para `Observer::update()`

```
TEST(ObserverTest, NeverUpdateObserver) {
  MockObserver observer;
  EXPECT_CALL(observer, update()).Times(0);

  Observable observable;
  observable.attach(&observer);
}
```

La prueba pasa. Ahora comprobemos que funciona también para dos observadores:

Listado 24.16: Prueba para notificación de dos observadores

```
TEST(ObserverTest, TwoObserver) {
  MockObserver observer1, observer2;
  EXPECT_CALL(observer1, update());
  EXPECT_CALL(observer2, update());

  Observable observable;
  observable.attach(&observer1);
  observable.attach(&observer2);

  observable.notify();
}
```

Y ejecutamos la prueba:

```
$ make test
Running main() from gmock_main.cc
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from ObserverTest
[ RUN      ] ObserverTest.UpdateObserver
[       OK ] ObserverTest.UpdateObserver (0 ms)
[ RUN      ] ObserverTest.NeverUpdateObserver
[       OK ] ObserverTest.NeverUpdateObserver (0 ms)
[ RUN      ] ObserverTest.TwoObserver
observable-tests.cc:30: Failure
```

```
12 Actual function call count doesn't match EXPECT_CALL(observer2, update
       ()).
13           Expected: to be called once
14             Actual: never called - unsatisfied and active
15 [  FAILED  ] ObserverTest.TwoObserver (0 ms)
16 [----------] 3 tests from ObserverTest (1 ms total)
17
18 [----------] Global test environment tear-down
19 [==========] 3 tests from 1 test case ran. (1 ms total)
20 [  PASSED  ] 2 tests.
21 [  FAILED  ] 1 test, listed below:
22 [  FAILED  ] ObserverTest.TwoObserver
23
24  1 FAILED TEST
```

Y la segunda expectativa falla (línea) y nos la muestra en consola:

```
Actual function call count doesn t match EXPECT_CALL(observer2, update
    ()).
```

Implementemos `notify()` para recorrer todos los observadores:

Listado 24.17: Patrón observador con TDD: `observable.cc`

```
1 void
2 Observable::notify(void) {
3   std::for_each(observers.begin(), observers.end(),
4         std::mem_fun(&Observer::update));
5 }
```

Ejecutamos de nuevo la prueba:

```
1  $ make test
2  Running main() from gmock_main.cc
3  [==========] Running 3 tests from 1 test case.
4  [----------] Global test environment set-up.
5  [----------] 3 tests from ObserverTest
6  [ RUN      ] ObserverTest.UpdateObserver
7  [       OK ] ObserverTest.UpdateObserver (0 ms)
8  [ RUN      ] ObserverTest.NeverUpdateObserver
9  [       OK ] ObserverTest.NeverUpdateObserver (0 ms)
10 [ RUN      ] ObserverTest.TwoObserver
11 [       OK ] ObserverTest.TwoObserver (0 ms)
12 [----------] 3 tests from ObserverTest (0 ms total)
13
14 [----------] Global test environment tear-down
15 [==========] 3 tests from 1 test case ran. (0 ms total)
16 [  PASSED  ] 3 tests.
```

Todo correcto, aunque sería conveniente una prueba adicional para un mayor número de observadores registrados. También podríamos comprobar que los observadores des-registrados (*detached*) efectivamente no son invocados, etc.

Aunque los ejemplos son sencillos, es fácil ver la dinámica de TDD.

24.8. Limitaciones

Hay ciertos aspectos importantes para la aplicación en los que TDD, y el *testing* en general, tienen una utilidad limitada (al menos hoy en día). Las pruebas permiten comprobar fácilmente aspectos funcionales, pero es complejo comprobar requisitos no funcionales.

«Los tests no pueden probar la ausencia de fallos,
sólo su existencia»

Kent Beck

Puede ser complicado probar rendimiento, fiabilidad, tiempo de respuesta y otros aspectos importantes, al menos escribiendo las pruebas primero. TDD tampoco ayuda a diseñar cuestiones de carácter general como la arquitectura de la aplicación, la seguridad, la accesibilidad, el modelo de persistencia, etc. Se dice que estos detalles de diseño «no emergen» de las pruebas.

Respecto al desarrollo de videojuegos, TDD no se adapta bien al diseño y prueba de la concurrencia, y en particular es complejo probar todo lo relacionado con la representación gráfica e interacción con el usuario. A pesar de ello, los métodos ágiles también están causando un importante impacto en el desarrollo de videojuegos y la informática gráfica en general. Cada día aparecen nuevos frameworks y herramientas que hacen posible probar de forma sencilla cosas que antes se consideraban inviables.

Capítulo 25

Empaquetado y distribución

Francisco Moya Fernández
David Villa Alises

Cada desarrollador de videojuegos tiene sus propios motivos para invertir su tiempo y esfuerzo en la elaboración de un producto tan complejo. Algunos programan juegos de forma altruista como carta de presentación o como contribución a la sociedad. Pero la mayoría de los desarrolladores de videojuegos trabajan para ver recompensado su esfuerzo de alguna forma, ya sea con donaciones, publicidad empotrada en el juego, o directamente cobrando por su distribución, o por las actualizaciones, o por nuevos niveles. En todos estos casos hay un factor común, el desarrollador quiere que el juego sea usado por la mayor cantidad posible de usuarios.

En capítulos anteriores hemos visto diferencias entre MS Windows y GNU/Linux desde el punto de vista de programación (plugins). En general son diferencias relativamente pequeñas, es perfectamente posible programar el videojuego para que se compile sin problemas en cualquiera de estas plataformas.

Pero el videojuego no solo tiene que compilarse, tiene que llegar al usuario de alguna forma que le permita instalarlo sin dificultad. En este capítulo veremos precisamente eso, la construcción de paquetes instalables para algunas plataformas (MS Windows y GNU/Linux).

A diferencia del resto de actividades de desarrollo los sistemas de empaquetado de software son completamente diferentes de unas plataformas a otras, e incluso hay alternativas muy diferentes en una misma plataforma. Nos centraremos en las que consideramos con mayor proyección.

25.1. Empaquetado y distribución en Windows

Existe una amplísima variedad de herramientas para el empaquetado en Microsoft Windows, y ninguna aglutina todas las ventajas[1]. En Microsoft Visual Studio anteriores a la versión 2012 (opción no disponible en la versión *Express*) era posible crear instaladores utilizando un tipo de proyecto especial (Visual Studio Setup Project, *vdproj*). Sin embargo esta opción, que ya había sido marcada como obsoleta en VS2010 terminó de eliminarse en la versión 2012. Para reemplazar esta funcionalidad Microsoft propone dos alternativas:

- **InstaShield 2010 Limited Edition (*ISLE*)**. Consiste en una versión restringida de la popular herramienta de generación de instaladores, actualmente propiedad de Flexera Software LLC. Esta edición limitada se puede descargar de forma gratuita previo registro, y se instala como un plugin de Visual Studio, pero limita las características del producto[2].
- **Windows Installer XML Toolset (*WIX*)**. Se trata de un conjunto de herramientas desarrolladas como un proyecto de software libre bajo la *Microsoft Reciprocal License (Ms-RL*. Fue inicialmente desarrollado por Microsoft y actualmente mantenido por un grupo de programadores que incluye a algunos de los desarrolladores de Visual Studio (por ejemplo, el líder del proyecto WIX, Rob Mensching). También existe un plugin para Visual Studio (*Votive*) para simplificar su uso.

En esta sección describiremos la segunda de las opciones. Microsoft utiliza WIX para desarrollar sus propios instaladores de Visual Studio, Office, o SQL Server. A diferencia de ISLE, WIX tiene un conjunto de características muy rico que va en progresivo aumento.

25.1.1. Creación de un paquete básico

Para ilustrar el funcionamiento de WIX utilizaremos un ejemplo sencillo de OGRE que simplemente muestra al personaje Sinbad de forma estática y permite el movimiento con cursores y ratón. Sin embargo empaquetaremos todos los componentes de OGRE para que la estructura del paquete pueda ser reutilizada en desarrollos más complejos. El código fuente del ejemplo puede obtenerse de `https://bitbucket.org/arco_group/ogre-hello/downloads/ogre-hello-0.1.tar.gz`.

Pre-requisitos

Antes de empezar el empaquetado tenemos que tener disponible una versión del proyecto correctamente compilada en MS Windows y probada. Puede utilizarse el conjunto de compiladores de GNU para Windows (MinGW) o Visual Studio. No es importante para esta sección cómo se generan los ejecutables.

En la documentación en línea de OGRE 1.8.1 se explica cómo construir proyectos con el SDK como proyecto de Visual Studio 2010 Express (Ver `http://www.ogre3d.org/tikiwiki/tiki-index.php`, apartado *Installing the Ogre SDK*). Es preciso puntualizar que no es necesario copiar el ejecutable al directorio de binarios de OGRE. Tan solo hay que garantizar que:

[1] Una comparativa simple con un subconjunto de las alternativas disponibles puede verse en `http://en.wikipedia.org/wiki/List_of_installation_software`.

[2] Ver `http://msdn.microsoft.com/en-us/library/ee721500(v=vs.100).aspx`

- Las bibliotecas dinámicas de OGRE (*.DLL) están en la ruta de búsqueda (típicamente en el mismo directorio de la aplicación).
- El directorio donde se meten los plugins de OGRE (típicamente el mismo de la aplicación) está correctamente consignado en el archivo de configuración correspondiente (`plugins.cfg`).
- El archivo de configuración de los recursos (`resources.cfg`) incluye referencias relativas al directorio de instalación de la aplicación.

En el mundo de las aplicaciones para Microsoft Windows es frecuente distribuir el software con la mayoría de las dependencias. Así, por ejemplo, cada juego realizado con OGRE tendría su propia versión de las DLL de OGRE. Esto tiene puntos positivos y negativos:

- El software ha sido realizado con una versión particular de OGRE. Si se utilizara cualquier otra es posible que se expusieran errores que no han sido detectados en el control de calidad. Distribuir OGRE con el propio juego permite que convivan múltiples versiones sin afectarse.
- Incluir en cada juego un conjunto de bibliotecas incrementa innecesariamente las necesidades de disco. Si 50 juegos tuvieran su propia copia particular de OGRE estaríamos ocupando casi 1GB más de lo necesario. En cualquier caso, al precio actual del almacenamiento esto tiene poca importancia. En la práctica real tiene aún menos importancia porque los fabricantes de videojuegos suelen desarrollar plataformas más o menos genéricas con los ejecutables, que se personalizan con scripts, plugins y archivos de medios. Solo el paquete de la plataforma tendrá una copia de las bibliotecas de OGRE.
- Los problemas arreglados en OGRE no se ven automáticamente arreglados en las aplicaciones que usan OGRE. No es posible actualizar OGRE en todos los paquetes que lo usan a la vez, es preciso actualizar cada uno de los juegos de forma independiente.
- Por tanto cada distribuidor de videojuegos se convierte en el responsable de actualizar todos los componentes de sus aplicaciones de la forma más diligente posible. Esto es también una oportunidad para crear ventajas competitivas ofreciendo un mejor servicio.

MS Windows Installer

Desde la aparición de Office 2000 Microsoft intenta estandarizar la instalación de software en los sistemas Windows alrededor de un software genérico de gestión de paquetes de software denominado Windows Installer. Además de distribuirlo dentro de Office 2000, Microsoft preparó un componente redistribuible que contenía Windows Installer y lo incluyó en todas las versiones posteriores de sus sistemas operativos. Se trata de un software de relativo bajo nivel que gestiona la base de datos de paquetes, y define el formato y estructura de los paquetes y de las actualizaciones. Incluye todo el software necesario para instalar, configurar, desinstalar, actualizar, reparar y analizar el software de un equipo.

Hasta entonces la única alternativa disponible era la distribución de un programa independiente en cada distribución de software (típicamente `setup.exe`) que debía ser ejecutado con privilegios de administrador. Este programa era el responsable de las tareas de instalación de los archivos (archivos `cab`), la configuración del registro o de los usuarios, etc. Al instalar el producto se solía instalar otro programa complementario (típicamente `uninstall.exe`) que permitía deshacer el proceso de instalación el producto.

Con Windows Installer toda la información pasa a estar contenida en un único archivo, el paquete de instalación, con extensión `msi`. También pueden instalarse paquetes de parches (con extensión `msp`).

Instalación mínima con WIX

Windows Installation XML Toolset es el primer proyecto liberado con una licencia libre por Microsoft en 2004. Inicialmente fue programado por Rob Mensching con el objetivo de facilitar la creación de archivos `msi` y `msp` sin necesidad de utilizar una interfaz gráfica.

WIX utiliza una aproximación puramente declarativa. En un archivo XML (con extensión `wxs`) se describe el producto y dónde se pueden encontrar cada uno de sus elementos. Este archivo es posteriormente compilado mediante la herramienta `candle.exe` para analizar su consistencia interna y generar un archivo intermedio, con extensión `wixobj`. Estos archivos intermedios siguen siendo XML aunque su contenido no está pensado para ser editado o leído por seres humanos.

Uno o varios archivos `wixobj` pueden ser procesados por otra herramienta, denominada `light.exe` para generar el archivo `msi` definitivo.

Este flujo de trabajo, muy similar al de la compilación de programas, es el más simple de cuantos permite WIX, pero también va a ser el más frecuente. Además de esto, WIX incluye herramientas para multitud de operaciones de gran utilidad en proyectos grandes.

Preparación del archivo XML

WIX sigue la filosofía de que el paquete de distribución es parte del desarrollo del programa. Por tanto el archivo `wxs` que describe el paquete debe escribirse de forma incremental junto al resto del programa.

Desgraciadamente en muchas ocasiones nos encontramos con que la tarea de empaquetado no se ha considerado durante el desarrollo del proyecto. Ya sea porque se ha utilizado otra plataforma para el desarrollo del videojuego, o por simple desidia, lo cierto es que frecuentemente llegamos al final del proceso de desarrollo sin ni siquiera habernos planteado la construcción del paquete. También en esos casos WIX aporta una solución, mediante la herramienta `heat.exe`. Esta herramienta puede construir fragmentos del archivo `wxs` mediante el análisis de directorios o archivos que van a incluirse en el paquete.

Una de las formas más sencillas de generar un paquete instalable es instalar los binarios y los archivos de medios necesarios durante la ejecución en un directorio independiente. El archivo generado no especialmente legible ni siquiera está completo, pero puede usarse como punto de partida.

Por ejemplo, si un programa instala todos los ejecutables y archivos auxiliares en el subdirectorio `bin` podemos generar el archivo `wxs` inicial con:

```
> heat.exe dir bin -out nombre-del-juego.wxs
```

Sin embargo, para mayor legibilidad en este primer ejemplo vamos a construir el archivo desde cero. Volvamos al ejemplo `ogre-hello`. Una vez generados los ejecutables en modo *Release* tenemos un subdirectorio `Release` que contiene todos los ejecutables, en este caso `OgreHello.exe`. Por sí solo no puede ejecutarse,

necesita las bibliotecas de OGRE, los archivos del subdirectorio `media` y los archivos de configuración `resources.cfg` y `plugins.cfg` que se incluyen con el código fuente. Es bastante habitual copiar las DLL al subdirectorio `Release` para poder probar el ejemplo.

La estructura del archivo `wxs` es la siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<Wix xmlns='http://schemas.microsoft.com/wix/2006/wi'>
  <Product Name='OgreHello 1.0'
           Id='06d99eb2-5f61-4753-a6fb-ba90166119cf'
           UpgradeCode='36a85162-a254-44b7-af91-44416d1c1935'
           Language='3082' Codepage='1252'
           Version='1.0.0' Manufacturer='UCLM'>

    <!-- aqui va la descripcion de los componentes del producto -->

  </Product>
</Wix>
```

La etiqueta *Wix* se usa para envolver toda la descripción del instalador. Dentro de ella debe haber un *Product* que describe el contenido del archivo `msi`. Todos los productos tienen los atributos `Id` y `UpgradeCode` que contienen cada uno un GUID (*Globally Unique Identifier*). Un GUID (o UUID) es un número largo de 128 bits, que puede generarse de manera que sea muy poco probable que haya otro igual en ningún otro sitio. Se utilizan por tanto para identificar de manera unívoca. En este caso se identifica el producto y el código de actualización.

Todas las versiones del producto tienen el mismo `Id`, mientras que para cambio de *major version* se genera otro nuevo GUID para `UpgradeCode` (al pasar de 1.x a 2.x, de 2.x a 3.x, etc.). El `UpgradeCode` es utilizado por *Windows Installer* para detectar cuando debe eliminar la versión vieja para reemplazarla por la nueva o simplemente reemplazar determinados archivos. Para generar un GUID nuevo puede utilizarse, por ejemplo, un generador de GUIDs en línea, como `http://www.guidgenerator.com/`.

Los códigos numéricos de *Language* y *Codepage* se corresponden a los valores indicados por microsoft en `http://msdn.microsoft.com/en-us/library/Aa369771.aspx`. En nuestro caso el idioma elegido es el español de España y la página de códigos es la correspondiente al alfabeto latino.

Dentro de un producto pueden declararse multitud de aspectos sobre el instalador. Lo primero es definir el paquete MSI que se va a generar:

```
    <Package Id='*'
             Description='Instalador de OgreHello 0.1'
             Manufacturer='UCLM'
             InstallerVersion='100'
             Languages='3082'
             Compressed='yes'
             SummaryCodepage='1252' />
```

Cada nuevo paquete generado tendrá su propio GUID. Por este motivo WIX permite simplificar la construcción con el código especial `*` que indica que debe ser auto-generado por el compilador. El atributo `InstallerVersion` indica la versión mínima de *Windows Installer* requerida para poder instalar el paquete. Si no se implementan aspectos avanzados, siempre será 100, que corresponde a la versión 1.0.

A continuación podemos declarar los archivos comprimidos que contiene el instalador. Se trata de archivos `cab` que son incluidos dentro del archivo `msi`. Habitualmente, dada la capacidad de los medios actuales solo es necesario incluir un archivo `cab`.

```
1     <Media Id='1' Cabinet='OgreHello.cab' EmbedCab='yes'/>
```

Ya solo queda describir la estructura de directorios que debe tener el producto. Se trata de una estructura jerárquica construida desde el directorio raíz, que siempre debe tener un atributo `Id` con valor *TARGETDIR* y un atributo `Name` con el valor *SourceDir*.

```
1  <Directory Id='TARGETDIR' Name='SourceDir'>
2    <Directory Id='ProgramFilesFolder' Name='PFiles'>
3      <Directory Id='INSTALLDIR' Name='OgreHello 1.0'>
4        <Directory Id='MediaDir' Name='media'/>
5        <Directory Id="ProgramMenuFolder" Name="Programs">
6          <Directory Id="ProgramMenuDir" Name="OgreHello 1.0"/>
7        </Directory>
8      </Directory>
9    </Directory>
10 </Directory>
```

En este caso hemos creado un directorio `OgreHello 1.0` dentro del directorio estándar para los archivos de programa (normalmente `C:\Program Files`). Dentro de este directorio hemos hecho un subdirectorio `media` que contendrá los archivos de medios (recursos).

Ahora podemos añadir componentes dentro de estos directorios. Cada componente es un conjunto de archivos muy fuertemente relacionado, hasta el punto de que no tiene sentido actualizar uno sin actualizar los demás. En general se tiende hacia componentes lo más pequeños posibles (un solo archivo), con objeto de que se puedan hacer parches más pequeños. Por ejemplo, el ejecutable principal es un componente, pero por simplicidad añadiremos también los archivos de configuración.

```
1  <DirectoryRef Id='INSTALLDIR' FileSource='.'>
2    <Component Id='MainExecutable' Guid='1e71f142-c7cd-4525-980b-78
         ebcafedeb1'>
3      <File Name='OgreHello.exe' KeyPath='yes'>
4        <Shortcut Id='startmenuOgreHello' Directory="ProgramMenuDir"
             Name="OgreHello 1.0"
5                  WorkingDirectory='INSTALLDIR' Icon='OgreHello.exe'
                       IconIndex='0' Advertise='yes' />
6      </File>
7      <File Name='plugins.cfg'/>
8      <File Name='resources.cfg'/>
9    </Component>
10 </DirectoryRef>
```

Cada componente tiene también un GUID que lo identifica. En este caso contiene tres archivos, el ejecutable y los dos archivos de configuración. Además, para el ejecutable creamos también un atajo en el menú de inicio de Windows.

El atributo `KeyPath` de los archivos se pone a *yes* solamente para un archivo dentro del componente. Este archivo será utilizado por *Windows Installer* para identificar si el componente está previamente instalado.

Para simplificar el resto del paquete vamos a meter todas las bibliotecas de OGRE en un único componente. En un caso real probablemente convendría dividirlo para permitir parches más pequeños en caso de que no afecten a todas las bibliotecas de OGRE.

```
<DirectoryRef Id='INSTALLDIR' FileSource='$(env.OGRE_HOME)\bin\
    release'>
  <Component Id='OgreDLLs' Guid='373f56f3-82c8-4c94-a0f6-
      d9be98d8d4af'>
    <File Name='OgreMain.dll' KeyPath='yes'/>
    <File Name='OgrePaging.dll'/>
    <File Name='OgreProperty.dll'/>
    <File Name='OgreRTShaderSystem.dll'/>
    <File Name='OgreTerrain.dll'/>
    <File Name='cg.dll'/>
    <File Name='OIS.dll'/>
    <File Name='Plugin_BSPSceneManager.dll'/>
    <File Name='Plugin_CgProgramManager.dll'/>
    <File Name='Plugin_OctreeSceneManager.dll'/>
    <File Name='Plugin_OctreeZone.dll'/>
    <File Name='Plugin_ParticleFX.dll'/>
    <File Name='Plugin_PCZSceneManager.dll'/>
    <File Name='RenderSystem_Direct3D9.dll'/>
    <File Name='RenderSystem_GL.dll'/>
  </Component>
</DirectoryRef>
```

Debe observarse el uso del atributo `FileSource` para configurar la fuente de los archivos a partir del valor de la variable de entorno `OGRE_HOME`. Esta variable contiene el directorio de instalación del SDK de OGRE si se han seguido los pasos indicados en la documentación.

A continuación queda añadir los archivos de medios.

```
<DirectoryRef Id='MediaDir' FileSource='media'>
  <Component Id='MediaFiles' Guid='9088eac3-9a72-4942-ba5e-28
      d870c90c36'>
    <File Name='Sinbad.mesh' KeyPath='yes'/>
    <File Name='Sinbad.material'/>
    <File Name='sinbad_body.tga'/>
    <File Name='sinbad_clothes.tga'/>
    <File Name='sinbad_sword.tga'/>
  </Component>
</DirectoryRef>
```

Un último componente nos permitirá borrar la carpeta `OgreHello 1.0` del menú de inicio en caso de desinstalación.

```
<DirectoryRef Id='ProgramMenuDir'>
  <Component Id="ProgramMenuDir" Guid="b16ffa2a-d978-4832-a5f2
      -01005e59853c">
    <RemoveFolder Id='ProgramMenuDir' On='uninstall' />
    <RegistryValue Root='HKCU' Key='Software\[Manufacturer]\[
        ProductName]' Type='string' Value='' KeyPath='yes' />
  </Component>
</DirectoryRef>
```

Cada producto software puede tener un conjunto de características que se instalan obligatoriamente o bien según la selección del usuario. En el caso más simple el paquete contiene una sola de estas características que instala todos los componentes que hemos definido anteriormente.

```
<Feature Id='Complete' Level='1'>
  <ComponentRef Id='MainExecutable' />
  <ComponentRef Id='OgreDLLs' />
  <ComponentRef Id='MediaFiles' />
  <ComponentRef Id='ProgramMenuDir' />
</Feature>
```

Como colofón podemos definir el icono de la aplicación. Normalmente se incluye dentro del mismo ejecutable, por lo que no es necesario añadir archivos nuevos.

```
<Icon Id="OgreHello.exe" SourceFile="OgreHello.exe" />
```

Construcción del paquete

Para construir el paquete es preciso ejecutar `candle.exe` para generar el archivo `wixobj` y posteriormente `light.exe` para generar el archivo `msi`. Por ejemplo, suponiendo que el archivo `wxs` está en el subdirectorio `wix` del proyecto y que los binarios están compilados en el subdirectorio `release` deberemos ejecutar:

```
> cd release
> candle.exe ..\wix\ogre-hello.wxs
> light.exe ogre-hello.wixobj
```

Si todo va correctamente, en el directorio `release` estará disponible el archivo `ògre-hello.msi` con el paquete de instalación.

Un doble click en el explorador es todo lo que se necesita para instalarlo. Alternativamente podemos usar la herramienta `msiexec` de *Windows Installer*:

```
> msiexec /i ogre-hello.msi
```

25.1.2. Interacción con el usuario

Hasta ahora los paquetes que hemos construido no tienen ningún tipo de interacción con el usuario. WIX permite añadir fácilmente todo tipo de interacción con el usuario para personalizar la instalación.

Para activar estas características es necesario ejecutar `light.exe` con la extensión *WixUIExtension*:

```
> light.exe -ext WixUIExtension ogre-hello.wixobj
```

Por ejemplo, para añadir un formulario que pregunta al usuario dónde se desea instalar tan solo tenemos que cambiar la sección *Feature* de este modo:

```
<UIRef Id="WixUI_Mondo" />
<UIRef Id="WixUI_ErrorProgressText" />
<Feature Id='Complete' Level='1'
         Title='OgreHello 1.0'
         Description='Componentes de OgreHello 1.0.'
         Display='expand'
         ConfigurableDirectory='INSTALLDIR'>
  <ComponentRef Id='MainExecutable' />
  <ComponentRef Id='OgreDLLs' />
  <ComponentRef Id='MediaFiles' />
  <ComponentRef Id='ProgramMenuDir' />
```

```
12 </Feature>
```

Tan solo hemos añadido dos referencias externas y algunos atributos, como `Title`, `Description`, `Display` y `ConfigurableDirectory`. El resultado es el que se muestra en la figura.

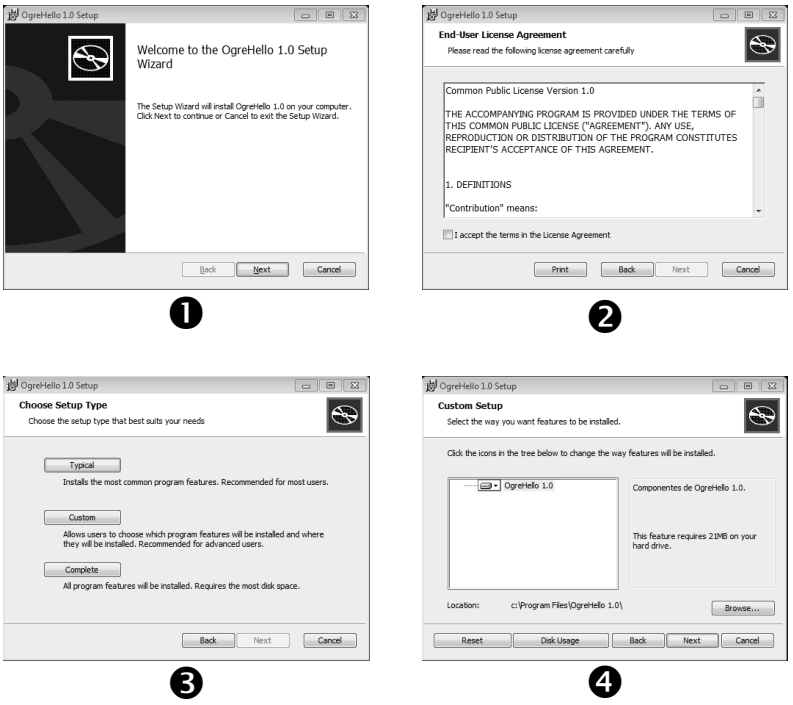


Figura 25.1: Ejemplo de diálogos generados por WIX.

25.1.3. Otras características

WIX es un producto complejo, con multitud de características avanzadas. Es posible crear bases de datos, replicar sitios web, instalar pre-requisitos, añadir o modificar permisos, alterar las reglas del *firewall*, etc. La utilización de XML permite que se puedan desarrollar paquetes modulares, con componentes compartidos por otros paquetes. Además permite crear con facilidad paquetes con parches, que solo alteran lo que ha sido modificado.

WIX dispone de documentación razonablemente completa, existe un excelente tutorial y ya se ha publicado un libro. Sin embargo las mejoras en WIX a veces van más rápido que la documentación de Internet. A menudo se leen comentarios acerca de la imposibilidad de crear paquetes con dependencias externas. Sin embargo los *bindles* de WIX permiten instalaciones encadenadas, incluso desde fuentes en red.

Recomendamos la consulta periódica de su página web `http://wixtoolset.org`.

25.2. Empaquetado y distribución en GNU/Linux

Debian GNU/Linux es una de las distribuciones de software libre más veteranas y de mayor calidad que existen actualmente. Debian se distingue principalmente por las siguientes características:

- No depende de ninguna empresa u organismo privado. Está completamente desarrollada y mantenida por voluntarios.
- Es libre y universal. El contrato social [3] de Debian es una declaración de valores y principios que recoge su objetivo y una definición detallada de las licencias libres que acoge.
- Asume un fuerte compromiso con la calidad, que se ve reflejado en una estricta y bien definida política[4] para incorporación de nuevo software y liberación de versiones.
- Soporta 12 arquitecturas diferentes, con varios núcleos en algunas de ellas.
- Un sofisticado sistema de paquetería (`.deb`), y herramientas para su gestión y actualización mediante APT (Advanced Packaging Tool). Probablemente ésta es la razón más importante por la que existe tal cantidad de distribuciones derivadas activas (unas 140) siendo Ubuntu la más popular.

Por estos motivos, `.deb` es probablemente el formato de paquete más adecuado para distribuir nuestro software en plataformas GNU/Linux.

El sistema de paquetería de Debian está definido por una serie de normas y procedimientos recogidos también en la Debian Policy. Uno de estos procedimientos especifica las habilidades que deben tener los desarrolladores (más conocidos como Debian Developers o DD). Solo los DD tiene permiso para subir nuevo software a los repositorios de Debian. Sin embargo, cualquier persona puede convertirse en DD si cumple una serie de exámenes. Es un proceso largo y muy burocrático, puesto que aparte de las habilidades técnicas requiere demostrar conocimientos aplicados de propiedad intelectual e industrial, y de licencias de software.

Afortunadamente, no es necesario realizar este proceso para colaborar con Debian. Es posible ser mantenedor (*maintainer*) de paquetes propios o ajenos. Los mantenedores realizan un trabajo similar, pero requieren la supervisión de un DD (un *sponsor*) para que su trabajo sea incluido en Debian de forma oficial.

Para aprender a empaquetar nuevo software o mantener paquetes existentes existe un documento llamado «Guía del nuevo mantenedor de Debian»[5]. Tanto la política como la guía del mantenedor explican como crear y mantener muchos tipos de paquetes: script de shell, programas de lenguajes interpretados, paquetes que dan lugar a un solo paquete binario, a múltiples, librerías dinámicas, módulos del kernel y muchos otros.

Esta sección se centra en describir el proceso para crear un paquete binario, que contendrá al menos un ejecutable ELF resultado de compilar un programa C++, además de ficheros auxiliares, que es el caso más habitual para distribuir un juego sencillo.

[3] `http://www.debian.org/social_contract.es.html`
[4] `http://www.debian.org/doc/debian-policy/`
[5] `http://www.debian.org/doc/manuals/maint-guide/index.en.html`

Se habla de un «paquete oficial» cuando está disponible a través de las copias espejo (*mirrors*) oficiales de Debian. Pero también es posible crear un paquete Debian correcto, pero que vamos a distribuir por nuestros propios medios (no-oficial). Aunque construir un paquete oficial es más complejo y requiere considerar más detalles, será ése el caso que abordemos aquí, puesto que siendo capaces de manejar un paquete oficial, se es capaz obviamente de manejar uno no oficial.

25.2.1. Pidiendo un paquete

Lo primero que debes averiguar antes de empezar es comprobar si el paquete existe previamente. Si no es el caso, debes comprobar si ya hay alguien que lo haya solicitado o que ya esté trabajando en ello. Puedes empaquetar cualquier programa que permita su distribución conforme a las directrices DFSG (Debian Free Software Guidelines) incluso sin pedir permiso al autor. Eso significa que aunque estés interesado en empaquetar un programa propio, podría ocurrir que ya hubiera alguien haciéndolo, siempre claro que tu programa esté disponible de algún modo y tenga una licencia libre.

Cualquiera puede solicitar que se empaquete un programa. Estas solicitud (y otras operaciones) se hacen reportando un error (*bug*) a un paquete especial llamado WNPP (Work-Needing and Prospective Packages) mediante el sistema DBTS (Debian Bug Tracking System), que está basado en comandos enviados a través de mensajes de correo electrónico. Vamos cuáles son esas operaciones:

- RFP (Request For Package) Corresponden con las solicitudes anteriormente mencionadas. Estas notificaciones aparecen en `http://www.debian.org/devel/wnpp/requested`. Las peticiones deberían incluir el nombre del programa, una breve descripción, el copyright y una URL para descarga.
- ITP (Intent to Package) Expresa la disposición del emisor de dicho reporte para empaquetar el programa por si mismo. Si vas a empaquetar un programa propio, debes enviar un ITP. Si ya existe, puedes escribir a su emisor para ofrecerle ayuda si tienes interés en que el paquete esté listo lo antes posible. Los paquetes en este estado apararecen en `http://www.debian.org/devel/wnpp/being_packaged`.
- RFH (Request For Help) Es una solicitud del mantenedor actual para que otros voluntarios de Debian le ayuden, ya se porque el programa es complejo o porque no dispone de tiempo para dedicarle. Aparecen listados en `http://www.debian.org/devel/wnpp/help_requested`.
- RFA (Request For Adoption) Indica que el mantenedor ya no quiere o no puede mantener el paquete, aunque se seguirá encargando hasta que aparezca un nuevo voluntario. Aparecen en `http://www.debian.org/devel/wnpp/rfa_bypackage`.
- O (Orphaned) Indica que el mantenedor actual ya no va a mantener el paquete. Se requiere un nuevo mantenedor lo antes posible. Se listan en `http://www.debian.org/devel/wnpp/orphaned`.

Para facilitar la creación de estos reportes (que se formatean como mensajes de correo) existe el programa `reportbug`. Se llama así porque en realidad este es el mecanismo para informar de cualquier problema en cualquier paquete oficial.

Asumiendo que ya has enviado el ITP puedes empezar a trabajar en la tarea de empaquetado propiamente dicha.

25.2.2. Obteniendo el fuente original

Lo habitual es que el autor (*upstream author*, usando la nomenclatura de Debian) proporcione los ficheros fuentes de su programa mediante un archivo `.tar.gz` (llamado a menudo *tarball*) o similar un su sitio web, o bien en un sistema de control de versiones público como github o bitbucket.

Vamos a empaquetar el programa `ogre-hello`, cuyo código fuente se puede descargar como vemos en el siguiente listado. Después se procede a descomprimir el archivo y a entrar en el directorio que aparece.

```
$ wget https://bitbucket.org/arco_group/ogre-hello/downloads/ogre-hello-0.1.tar.
    gz
$ tar xvfz ogre-hello-0.1.tar.gz
$ cd ogre-hello-0.1
ogre-hello-0.1$
```

25.2.3. Estructura básica

En este punto podemos utilizar el programa `dh_make` que nos ayuda a generar las plantillas de ficheros necesarios para conctruir el paquete Debian, pero antes de eso debemos crear dos variables de entorno para el nombre y dirección de correo del mantenedor:

```
DEBEMAIL="juan.nadie@gmail.com"
DEBFULLNAME="Juan Nadie"
export DEBEMAIL DEBFULLNAME
```

Estas variables son útiles para otras operaciones relacionadas con las construcción de paquetes, por lo que es buena idea añadirlas al fichero `$HOME/.bashrc` para futuras sesiones. Después de eso, ejecuta `dh_make`:

```
ogre-hello-0.1$ dh_make -f ../ogre-hello-0.1.tar.gz
Type of package: single binary, indep binary, multiple binary, library, kernel
    module, kernel patch?
 [s/i/m/l/k/n] s

Maintainer name  : Juan Nadie
Email-Address    : juan.nadie@gmail.com
Date             : Wed, 24 Apr 2013 12:59:40 +0200
Package Name     : ogre-hello
Version          : 0.1
License          : blank
Type of Package  : Single
Hit <enter> to confirm:
Done. Please edit the files in the debian/ subdirectory now. You should also
check that the ogre-hello Makefiles install into $DESTDIR and not in / .
```

manpages

Todo fichero ejecutable en el `PATH` debería tener su página de manual. Las páginas de manual están escritas en `groff`, pero es mucho más sencillo crearlas a partir de formatos más sencillos como SGML (Standard Generalized Markup Language), XML, asciidoc o reStructuredText y convertirlas en el momento de la construcción del paquete.

La ejecución de `dh_make` ha creado un subdirectorio llamado `debian` con muchos ficheros. Muchos de ellos son ejemplos (extensión `.ex`) para distintos fines: página de manual en distintos formatos, acciones durante la instalación o desinstalación del paquete, script para creación de un servicio en background (*daemon*), integración con `cron`, y muchos otros usos. Entre todos ellos hay cuatro ficheros esenciales que todo paquete Debian debe tener. A continuación de muestran y describen, una vez modificados según corresponde con el paquete en cuestión. Compárelos con los generados por `dh_make`.

debian/control

El fichero `debian/control` es una especie de *manifiesto* del paquete. Incluye información esencial para el sistema de gestión de paquetes. Veamos el fichero generado:

Listado 25.1: dh_make: `debian/control`

```
1  Source: ogre-hello
2  Section: unknown
3  Priority: extra
4  Maintainer: Juan Nadie <Juan.Nadie@gmail.com>
5  Build-Depends: debhelper (>= 8.0.0)
6  Standards-Version: 3.9.4
7  Homepage: <insert the upstream URL, if relevant>
8  #Vcs-Git: git://git.debian.org/collab-maint/ogre-hello.git
9  #Vcs-Browser: http://git.debian.org/?p=collab-maint/ogre-hello.git;
       a=summary
10
11 Package: ogre-hello
12 Architecture: any
13 Depends: ${shlibs:Depends}, ${misc:Depends}
14 Description: <insert up to 60 chars description>
15  <insert long description, indented with spaces>
```

Este fichero tiene dos secciones claramente diferenciadas. La sección `Source` (sólo puede haber una) describe el paquete *fuente* y la información necesaria para su construcción. Cada sección `Package` (puede haber varias aunque en este caso solo haya una) describe los paquetes binarios (los archivos `.deb`) resultado de la construcción del paquete fuente. Veamos el significado de cada uno de los campos del fichero:

Campos de la sección *fuente*:

Source
: Es el nombre del paquete fuente, normalmente el mismo nombre del programa tal como lo nombró el autor original (*upstream author*).

Section
: La categoría en la que se clasifica el paquete, dentro de una lista establecida, vea `http://packages.debian.org/unstable/`.

Priority
: La importancia que tiene el paquete para la instalación. Las categorías son: *required*, *important*, *standard*, *optional* y *extra*. Este paquete, al ser un juego y no causar ningún conflicto, es *optional*. Puede ver el significado de cada una de estas prioridades en `http://www.debian.org/doc/debian-policy/ch-archive.html`.

Maintainer
: Es el nombre y la dirección de correo electrónico de la persona que mantiene el paquete actualmente.

Build-Depends
: Es una lista de nombres de paquetes (opcionalmente con las versiones requeridas) que son necesarios para compilar el presente paquete. Estos paquetes deben estar instalados en el sistema (además del paquete `build-essential`) para poder construir los paquetes «binarios».

Standards-Version
: La versión más reciente de la *policy* que cumple el paquete.

Campos de la sección *binaria*:

Package
El nombre del paquete binario, que dará lugar al archivo `.deb`. Este nombre debe cumplir una serie de reglas que dependen del lenguaje en que está escrito el programa, su finalidad, etc.

Architecture
Indica en qué arquitecturas hardware puede compilar el programa. Aparte de las arquitecturas soportadas por Debian hay dos identificadores especiales:

- «all», indica que el mismo programa funciona en todas las plataformas. Normalmente se trata de programas escritos en lenguajes interpretados o bien ficheros multimedia, manuales, etc, que no requieren compilación. Se dice que son paquetes «independientes de arquitectura».
- «any», indica que el programa debe ser compilado pero está soportado en todas las arquitecturas.

Depends
Es una lista de los nombres de los paquetes necesarios para instalar el paquete y ejecutar los programas que contiene.

Debe quedar clara la diferencia entre el campo `Build-Depends` y el campo `Depends`. El primero contiene dependencias necesarias para la **construcción** del paquete, mientras que el segundo lista las dependencias para su **ejecución**. Es fácil comprobar la diferencia. Cuando se requieren librerías, `Build-Depends` contiene las versiones `-dev` (como `libogre-1.8-dev` en nuestro caso), que incluyen los ficheros de cabecera, mientras que `Depends` contiene la versión de usuario (`libogre-1.8.0`).

Después de nuestras modificaciones podría quedar del siguiente modo:

Listado 25.2: `debian/control` adaptado a `ogre-hello`

```
 1 ource: ogre-hello
 2 Section: games
 3 Priority: optional
 4 Maintainer: Juan Nadie <Juan.Nadie@gmail.com>
 5 Build-Depends: debhelper (>= 9.0.0), libogre-1.8-dev, libois-dev,
       libgl1-mesa-dev, quilt
 6 Standards-Version: 3.9.4
 7
 8 Package: ogre-hello
 9 Architecture: any
10 Depends: ${shlibs:Depends}, ${misc:Depends}, libogre-1.8.0
11 Description: minimal packaged Ogre example
12  minimal Ogre example to demostrate Debian packaging
```

debian/changelog

Contiene una descripción breve de los cambios que el mantenedor hace a los ficheros específicos del paquete (los que estamos describiendo). En este fichero no se describen los cambios que sufre el programa en sí; eso corresponde al autor del programa y normalmente estarán en un fichero `CHANGES` (o algo parecido) dentro del `.tgz` que descargamos.

El siguiente listado muestra el fichero `changelog` generado por `dh_make`:

Listado 25.3: debian/changelog para ogre-hello

```
ogre-hello (0.1-1) unstable; urgency=low

  * Initial release (Closes: #nnnn)  <nnnn is the bug number of your ITP>

-- Juan Nadie <Juan.Nadie@gmail.com>  Wed, 24 Apr 2013 12:59:40 +0200
```

Cada vez que el mantenedor haga un cambio debe crear una nueva entrada como esa al comienzo del fichero. Fíjese que la versión del paquete está formada por el número de versión del programa original más un guión y un número adicional que indica la *revisión* del paquete debian. Si el mantenedor hace cambios sobre el paquete manteniendo la misma versión del programa irá incrementando el número tras el guión. Sin embargo, cuando el mantenedor empaqueta una nueva versión del programa (supongamos la 0.2 en nuestro ejemplo) el número tras el guión vuelve a empezar desde 1.

Como se puede apreciar, la primera versión del paquete debería cerrar (solucionar) un bug existente. Ese bug (identificado por un número) corresponde al ITP que el mantenedor debió enviar antes de comentar con el proceso de empaquetado.

Cuando una nueva versión del paquete se sube a los repositorios oficiales, las sentencias `Closes` son procesadas automáticamente para cerrar los bugs notificados a través del DBTS. Puedes ver más detalles en `http://www.debian.org/doc/debian-policy/ch-source.html#s-dpkgchangelog`

debian/copyright

Este fichero debe contender toda la información sobre el autor del programa y las licencias utilizadas en cada una de sus partes (si es que son varias). También debería indicar quién es el autor y la licencia de los ficheros del paquete debian. El fichero generado por `dh_make` contiene algo similar a:

Listado 25.4: dh_make: debian/copyright

```
Format: http://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: ogre-hello
Source: <url://example.com>

Files: *
Copyright: <years> <put author's name and email here>
           <years> <likewise for another author>
License: <special license>
 <Put the license of the package here indented by 1 space>
 <This follows the format of Description: lines in control file>
 .
 <Including paragraphs>

# If you want to use GPL v2 or later for the /debian/* files use
# the following clauses, or change it to suit. Delete these two lines
Files: debian/*
Copyright: 2013 Juan Nadie <Juan.Nadie@gmail.com>
License: GPL-2+
 This package is free software; you can redistribute it and/or modify
```

```
 20  it under the terms of the GNU General Public License as published by
 21  the Free Software Foundation; either version 2 of the License, or
 22  (at your option) any later version.
 23  .
 24  This package is distributed in the hope that it will be useful,
 25  but WITHOUT ANY WARRANTY; without even the implied warranty of
 26  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 27  GNU General Public License for more details.
 28  .
 29  You should have received a copy of the GNU General Public License
 30  along with this program. If not, see <http://www.gnu.org/licenses/>
 31  .
 32  On Debian systems, the complete text of the GNU General
 33  Public License version 2 can be found in "/usr/share/common-licenses/GPL-2".
 34
 35 # Please also look if there are files or directories which have a
 36 # different copyright/license attached and list them here.
 37 # Please avoid to pick license terms that are more restrictive than the
 38 # packaged work, as it may make Debian's contributions unacceptable upstream.
```

El fichero debería contener una sección (que comienza con «`Files:`)» por cada autor y licencia involucrados en el programa. La sección «`Files: debian/*`» ya está completa y asume que el mantenedor va a utilizar la licencia GPL-2 y superior para los ficheros del paquete.

Debemos modificar este fichero para incluir las licencias específicas del programa que estamos empaquetando. Las secciones nuevas serían:

Listado 25.5: `debian/copyright` para `ogre-hello`

```
 1
 2 Format: http://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
 3 Upstream-Name: ogre-hello
 4 Source: http://www.ogre3d.org/tikiwiki/Sinbad+Model
 5
 6 Files: *
 7 Copyright: 2013 ogre.developers@ogre.com
 8 License: GPL-2
 9  This package is free software; you can redistribute it and/or modify
10  it under the terms of the GNU General Public License as published by
11  the Free Software Foundation; either version 2 of the License, or
12  (at your option) any later version.
13  .
14  This package is distributed in the hope that it will be useful,
15  but WITHOUT ANY WARRANTY; without even the implied warranty of
16  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17  GNU General Public License for more details.
18  .
19  You should have received a copy of the GNU General Public License
20  along with this program. If not, see <http://www.gnu.org/licenses/>
21  .
22  On Debian systems, the complete text of the GNU General
23  Public License version 2 can be found in "/usr/share/common-licenses/GPL-2".
24
25 Files: media/*
26 Copyright: 2009-2010  Zi Ye <omniter@gmail.com>
27 License:
28  This work is licensed under the Creative Commons Attribution-Share
29  Alike 3.0 Unported License.  To view a copy of this license, visit
30  http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to
31  Creative Commons, 171 Second Street, Suite 300, San Francisco,
32  California, 94105, USA.
33  .
34  This character is a gift to the OGRE community
35  (http://www.ogre3d.org).  You do not need to give credit to the
36  artist, but it would be appreciated. =)
```

debian/rules

Se trata de un fichero `Makefile` de GNU Make que debe incluir un serie de objetivos: clean, binary, binary-arch, binary-indep y build. Estos objetivos son ejecutados por el programa `dpkg-buildpackage` durante la construcción del paquete.

El listado muestra el fichero `rules` generado:

Listado 25.6: dh_make: `debian/rules`

```
 1 #!/usr/bin/make -f
 2 # -*- makefile -*-
 3 # Sample debian/rules that uses debhelper.
 4 # This file was originally written by Joey Hess and Craig Small.
 5 # As a special exception, when this file is copied by dh-make into a
 6 # dh-make output file, you may use that output file without restriction.
 7 # This special exception was added by Craig Small in version 0.37 of dh-make.
 8
 9 # Uncomment this to turn on verbose mode.
10 #export DH_VERBOSE=1
11
12 %:
13     dh $@ --with quilt
```

Por fortuna, actualmente el programa `debhelper` (o `dh`) hace la mayor parte del trabajo aplicando reglas por defecto para todos los objetivos. Solo es necesario sobreescribir (reglas `override_` dichos objetivos en el fichero `rules` si se requiere un comportamiento especial.

El listado anterior contiene una pequeña diferencia respecto al generado por `dh_make` que consiste en la adición del parámetro `-with quilt`. El programa `quilt` es una aplicación especializada en la gestión de parches de un modo muy cómodo.

25.2.4. Construcción del paquete

build-depends

Para construir el paquete debe tener instalados todos los paquetes listados en los campos `Build-Depends` y `Build-Depends-Indep`, además del paquete `build-essential`.

Una vez contamos con los ficheros básicos se puede proceder a una primera compilación del paquete. Para ello utilizamos el programa `dpkg-buildpackage`. El siguiente listado muestra el comando y parte del resultado. Se omiten algunas partes dado que la salida es bastante verbosa:

```
ogre-hello-0.1$ dpkg-buildpackage -us -us
dpkg-buildpackage: source package ogre-hello
dpkg-buildpackage: source version 0.1-1
dpkg-buildpackage: source changed by Juan Nadie <Juan.Nadie@gmail.com>
dpkg-buildpackage: host architecture amd64
 dpkg-source --before-build ogre-hello-0.1
dpkg-source: info: applying make-install
 fakeroot debian/rules clean
dh clean --with quilt
   dh_testdir
   dh_auto_clean
make[1]: Entering directory '/home/david/repos/ogre-hello/ogre-hello-0.1'
rm -f helloWorld *.log *.o *~
make[1]: Leaving directory '/home/david/repos/ogre-hello/ogre-hello-0.1'
   dh_quilt_unpatch
 dpkg-source -b ogre-hello-0.1
dpkg-source: info: using source format '3.0 (quilt)'
dpkg-source: info: applying make-install
dpkg-source: info: building ogre-hello using existing ./ogre-hello_0.1.orig.tar.
    gz
dpkg-source: info: building ogre-hello in ogre-hello_0.1-1.debian.tar.gz
dpkg-source: info: building ogre-hello in ogre-hello_0.1-1.dsc
 debian/rules build
dh build --with quilt
   [...]
   dh_testdir
```

```
make[1]: Entering directory '/home/david/repos/ogre-hello/ogre-hello-0.1'
g++ -I. -Wall -O2 -D_RELEASE `pkg-config --cflags OGRE`  `pkg-config --libs OGRE
     ` -lOIS -lstdc++  helloWorld.cpp   -o helloWorld
make[1]: Leaving directory '/home/david/repos/ogre-hello/ogre-hello-0.1'
   dh_auto_test
 fakeroot debian/rules binary
dh binary --with quilt
   [...]
   dh_md5sums
   dh_builddeb
dpkg-deb: building package 'ogre-hello' in '../ogre-hello_0.1-1_amd64.deb'.
 dpkg-genchanges  >../ogre-hello_0.1-1_amd64.changes
dpkg-genchanges: including full source code in upload
 dpkg-source --after-build ogre-hello-0.1
dpkg-buildpackage: full upload (original source is included)
```

Fíjese en los parámetros de la llamada: `-us -uc`. Piden que no se firmen digitalmente ni el paquete fuente ni el fichero `.changes` respectivamente.

Este proceso ha creado tres ficheros en el directorio padre:

`ogre-hello_0.1-1.dsc`
: Es una descripción del paquete fuente, con una serie de campos extraídos del fichero `debian/control` además de los *checksums* para los otros dos ficheros que forman el paquete fuente: el `.orig.tar.gz` y el `.debian.tar.gz`.

Listado 25.7: Fichero con la descripción del paquete `ogre-hello`

```
 1 Format: 3.0 (quilt)
 2 Source: ogre-hello
 3 Binary: ogre-hello
 4 Architecture: any
 5 Version: 0.1-1
 6 Maintainer: Juan Nadie <Juan.Nadie@gmail.com>
 7 Standards-Version: 3.9.4
 8 Build-Depends: debhelper (>= 9.0.0), libogre-1.8-dev, libois-dev,
       libgl1-mesa-dev, quilt
 9 Package-List:
10  ogre-hello deb games optional
11 Checksums-Sha1:
12  651807d3ca4d07a84e80eb2a20f4fe48eb986845 1188396 ogre-hello_0.1.orig.
        tar.gz
13  5e59ae17c6fd21a69573fc61bdd85bbb275ced68 1870 ogre-hello_0.1-1.debian.
        tar.gz
14 Checksums-Sha256:
15  48f09390131bb0ea66d91a1cd27fbe6ccc9b29a465159b6202f0f2dc38702b10
        1188396 ogre-hello_0.1.orig.tar.gz
16  79275174e5a2b358cd3b031db6dc46faca0a5865c1b22fcb778c6e05e594e4b9 1870
        ogre-hello_0.1-1.debian.tar.gz
17 Files:
18  4d5e668550d95dc0614435c1480a44e1 1188396 ogre-hello_0.1.orig.tar.gz
19  83cf823860cc404ece965d88b565e491 1870 ogre-hello_0.1-1.debian.tar.gz
```

`ogre-hello_0.1-1_amd64.changes`
: Este fichero es utilizado por el archivo (el repositorio) de paquetes de Debian y se utiliza en la subida (*upload*) de paquetes por parte de los DD (Debian Developer). Contiene *checksum* para los otros ficheros y, en una versión oficial, debería estar firmado digitalmente para asegurar que el paquete no ha sido manipulado por terceros.

`ogre-hello_0.1-1.debian.tar.gz`
: Es un archivo que contiene todos los cambios que el mantenedor del paquete ha hecho respecto al fuente del programa original, es decir, el directorio `debian` principalmente.

`ogre-hello_0.1-1_amd64.deb`
: . Es el paquete *binario* resultado de la compilación (en concreto para la arquitectura AMD-64) que puede ser instalado con `dpkg` o indirectamente con los gestores de paquetes `apt-get`, `aptitude` u otros.

Es posible descargar y construir el paquete binario (`.deb`) a partir de los ficheros `.dsc`, `.debian.tar.gz` y `.orig.tar.gz`. La forma más simple de hacer esto con un paquete disponible en el repositorio es el comando `apt-get -build source nombre-de-paquete`

.

Aunque aparentemente todo ha ido bien y se ha generado el fichero `.deb` vamos a comprobar que hay algunos problemas graves por resolver. La forma más eficaz de ver estos problemas en utilizar el programa `lintian`:

```
ogre-hello-0.1$ lintian ../ogre-hello\_0.1-1\_amd64.changes
W: ogre-hello: empty-binary-package
```

Este aviso indica que el paquete no contiene nada aparte de los ficheros aportados por el propio sistema de construcción. Veamos qué contiene realmente:

```
ogre-hello-0.1$ debc ../ogre-hello_0.1-1_amd64.deb
 new debian package, version 2.0.
 size 1192818 bytes: control archive=760 bytes.
     352 bytes,    10 lines      control
     836 bytes,    11 lines      md5sums
 Package: ogre-hello
 Version: 0.1-1
 Architecture: amd64
 Maintainer: Juan Nadie <Juan.Nadie@gmail.com>
 Installed-Size: 2814
 Depends: libc6 (>= 2.2.5), libgcc1 (>= 1:4.1.1), libogre-1.8.0, libois-1.3.0,
      libstdc++6 (>= 4.2.1)
 Section: games
 Priority: optional
 Description: minimal packaged Ogre example
  minimal Ogre example to demostrate Debian packaging

*** Contents:
drwxr-xr-x root/root         0 2013-04-29 14:05 ./
drwxr-xr-x root/root         0 2013-04-29 14:05 ./usr/
drwxr-xr-x root/root         0 2013-04-29 14:05 ./usr/share/
drwxr-xr-x root/root         0 2013-04-29 14:05 ./usr/share/doc/
drwxr-xr-x root/root         0 2013-04-29 14:05 ./usr/share/doc/ogre-hello/
-rw-r--r-- root/root      2348 2013-04-29 13:15 ./usr/share/doc/ogre-hello/
     copyright
-rw-r--r-- root/root       175 2013-04-29 13:15 ./usr/share/doc/ogre-hello/
     changelog.Debian.gz
```

Esto se debe a que no basta con compilar el programa, es necesario instalar los ficheros en el lugar adecuado. Esta es uno de los principales objetivos del empaquetado.

25.2.5. Parches: adaptación a Debian

Instalar el programa y sus ficheros asociados requiere en este caso modificar el fichero `Makefile`. El siguiente listado muestra esos cambios respetando el lugar adecuado según la política de Debian:

Listado 25.8: Parche para la instalación de `ogre-hello`

```
1 install: helloWorld
2      install -vd $(DESTDIR)/usr/games
3      install -v -m 444 helloWorld $(DESTDIR)/usr/games/ogre-hello
4      install -vd $(DESTDIR)/usr/share/games/ogre-hello/
5      install -v -m 444 *.cfg $(DESTDIR)/usr/share/games/ogre-hello/
6      install -vd $(DESTDIR)/usr/share/games/ogre-hello/media
7      install -v -m 444 media/*.tga $(DESTDIR)/usr/share/games/ogre-
           hello/media/
```

```
8       install -v -m 444 media/*.material $(DESTDIR)/usr/share/games/
            ogre-hello/media/
9       install -v -m 444 media/*.mesh $(DESTDIR)/usr/share/games/ogre-
            hello/media/
```

Es imporante destacar que toda la instación debe hacerse respecto a un directorio contenido en la variable `DESTDIR`. Se utiliza el programa `install` para crear los directorios necesarios y copiar cada fichero a sus lugar, indicando además los permisos de acceso y ejecución que correspondan.

Sin embargo, si tratamos de construir el programa utilizando la herramienta `dpkg-buildpackage` en este momento estado obtendremos un error:

```
dpkg-source: info: local changes detected, the modified files are:
 ogre-hello-0.1/makefile
dpkg-source: info: you can integrate the local changes with dpkg-source --commit
```

Se debe a que no está permitido modificar los ficheros extraídos del tarball del autor del programa. Es necesario crear un *parche*. Un parche es un fichero que contiene los cambios que es preciso realizar en otro fichero para lograr un resultado concreto, indicando el número de línea y otro contenido que ayuda a las herramientas a localizar el lugar correcto. Por suerte, el propio error nos ofrece una forma muy sencilla que crear este parche:

```
ogre-hello-0.1$ dpkg-source --commit
dpkg-source: info: local changes detected, the modified files are:
 ogre-hello-0.1/makefile
Enter the desired patch name: make-install
```

El programa `dpkg-source` pide un nombre para el parche (le damos `make-install`) y como resultado:

- Deja el fichero `makefile` tal como estaba antes de nuestro cambio.
- Crea el parche en `debian/patches/make-install`.
- Crea el fichero `debian/patches/series` que contendrá los nombres de todos los parches a aplicar (`make-install`) en este momento.

Estos parches serán aplicados por el programa `quilt`, que se invocará automáticamente al usar `dpkg-buildpackage`. Veamos las diferencias:

```
ogre-hello-0.1$ dpkg-buildpackage -us -us -rfakeroot
[...]
dh build --with quilt
   dh\_testdir
   dh\_quilt\_patch
File series fully applied, ends at patch make-install
[...]
install -vd /home/david/repos/ogre-hello/ogre-hello-0.1/debian/ogre-hello/usr/
    games
install: creating directory '/home/david/repos/ogre-hello/ogre-hello-0.1/debian/
    ogre-hello/usr'
install: creating directory '/home/david/repos/ogre-hello/ogre-hello-0.1/debian/
    ogre-hello/usr/games'
install -v -m 444 helloWorld /home/david/repos/ogre-hello/ogre-hello-0.1/debian/
    ogre-hello/usr/games/ogre-hello
[....]
```

Veamos qué problemas detecta `lintian`:

```
ogre-hello-0.1$ lintian ../ogre-hello_0.1-1_amd64.changes
W: ogre-hello: binary-without-manpage usr/games/ogre-hello
W: ogre-hello: hardening-no-relro usr/games/ogre-hello
```

Y el contenido del paquete binario:

```
ogre-hello-0.1$ debc ../ogre-hello\_0.1-1\_amd64.deb
[...]
*** Contents:
drwxr-xr-x root/root         0 2013-04-29 14:26 ./
drwxr-xr-x root/root         0 2013-04-29 14:26 ./usr/
drwxr-xr-x root/root         0 2013-04-29 14:26 ./usr/games/
-rwxr-xr-x root/root     37912 2013-04-29 14:26 ./usr/games/ogre-hello
drwxr-xr-x root/root         0 2013-04-29 14:26 ./usr/share/
drwxr-xr-x root/root         0 2013-04-29 14:26 ./usr/share/doc/
drwxr-xr-x root/root         0 2013-04-29 14:26 ./usr/share/doc/ogre-hello/
-rw-r--r-- root/root      2348 2013-04-29 13:15 ./usr/share/doc/ogre-hello/
    copyright
-rw-r--r-- root/root       175 2013-04-29 13:15 ./usr/share/doc/ogre-hello/
    changelog.Debian.gz
drwxr-xr-x root/root         0 2013-04-29 14:26 ./usr/share/games/
drwxr-xr-x root/root         0 2013-04-29 14:26 ./usr/share/games/ogre-hello/
-rw-r--r-- root/root        27 2013-04-29 14:26 ./usr/share/games/ogre-hello/
    resources.cfg
-rw-r--r-- root/root       225 2013-04-29 14:26 ./usr/share/games/ogre-hello/
    ogre.cfg
-rw-r--r-- root/root       112 2013-04-29 14:26 ./usr/share/games/ogre-hello/
    plugins.cfg
drwxr-xr-x root/root         0 2013-04-29 14:26 ./usr/share/games/ogre-hello/
    media/
-rw-r--r-- root/root      2519 2013-04-29 14:26 ./usr/share/games/ogre-hello/
    media/Sinbad.material
-rw-r--r-- root/root    786476 2013-04-29 14:26 ./usr/share/games/ogre-hello/
    media/sinbad\_body.tga
-rw-r--r-- root/root   1026978 2013-04-29 14:26 ./usr/share/games/ogre-hello/
    media/Sinbad.mesh
-rw-r--r-- root/root    196652 2013-04-29 14:26 ./usr/share/games/ogre-hello/
    media/sinbad\_sword.tga
-rw-r--r-- root/root    786476 2013-04-29 14:26 ./usr/share/games/ogre-hello/
    media/sinbad\_clothes.tga
```

Aunque ahora el paquete contiene los ficheros deseados y los instalará en su ruta correcta, pero aún tiene algunos problemas ya que el programa no estaba pensado para trabajar con esta estructura de ficheros:

- Los ficheros de configuración de ogre se buscan en directorio actual, pero queremos buscarlos en `/usr/share/games/ogre-hello`.

  Para solucionarlo, editamos el fichero `ExampleApplication.h` (línea 168) y, a continuación, asignamos el valor `/usr/share/games/ogre-hello/` a la variable `mResourcePath`. Después se ejecuta `dpkg-source -commit` y escribimos `resource-path` como nombre del parche.

- Los recursos gráficos se buscan en `media`, pero en la versión instalada deberían buscarse en `/usr/share/games/ogre-hello/media`. En este caso se debe editar la variable `FyleSystem` del fichero `media.cfg` dándole el valor `/usr/share/games/ogre-hello/media/`.

Con esto último tendremos un total de tres parches y el programa será funcional tras la instalación. Quedan aún dos problemas (no tan graves) por resolver según informa `lintian`. El primero (binary-without-manpage) indica que todo fichero ejecutable en el `PATH` debería tener una página de manual. El segundo (hardening-no-relro) indica que el programa debería estar compilado con determinadas opciones que evitan problemas comunes.

25.2.6. Actualización del paquete

Mantener un paquete no acaba con la construcción de un paquete correcto y funcional. Lo normal será que el autor del programa continúe mejorando su aplicación y liberando versiones nuevas (*releases*). Además, los usuarios de Debian pueden encontrar e informar de problemas en el paquete que también deben ser reparados. En ambos casos el mantenedor debe actualizar el paquete y crear una nueva versión (en el primer caso) o revisión (en el segundo caso).

El mantenedor puede tener conocimiento de una nueva versión del programa mediante una notificación de los usuarios al DBTS. Sin embargo, existe un método automático para lograrlo. El paquete puede contar con un fichero especial en `debian/watch` que contiene normalmente una URL con una expresión regular para localizar los ficheros fuente de todas las versiones que proporcione el autor. El siguiente listado muestra el fichero `watch` para el programa `ogre-hello`.

El fichero `watch` es procesado automáticamente por el sistema DEHS (Debian External Health Status) de Debian. Este sistema lleva el control de todos los paquetes de cada mantenedor y le permite comprobar fácilmente el estado de todos sus paquetes[6].

Listado 25.9: Fichero `watch` para `ogre-hello`

```
1 version=3
2 http://bitbucket.org/arco_group/ogre-hello/downloads/ogre-hello-(.*)\.tar\.gz
```

Obviamente esto resulta útil cuando mantenemos un programa de un tercero, pero también es interesante incluso aunque estemos empaquetando una aplicación propia. El programa `uscan` puede procesar este fichero y descargar automáticamente con los fuentes del programa. Incluso puede crear un enlace a él con el nombre `.orig.tar.gz` que cumple las normas de Debian.

```
ogre-hello-0.1$ uscan --verbose --download-current-version \
        --force-download --repack --destdir ..
-- Scanning for watchfiles in .
-- Found watchfile in ./debian
-- In debian/watch, processing watchfile line:
   http://bitbucket.org/arco_group/ogre-hello/downloads/ogre-hello
       -(.*)\.tar\.gz
-- Found the following matching hrefs:
     /arco_group/ogre-hello/downloads/ogre-hello-0.1.tar.gz
     /arco_group/ogre-hello/downloads/ogre-hello-0.1.tar.gz
Newest version on remote site is 0.1, local version is 0.1
 => Package is up to date
Newest version on remote site is 0.1, local version is 0.1
 => ogre-hello-0.1.tar.gz already in package directory
-- Scan finished
```

Las cuatro últimas líneas de la salida de `uscan` confirman que tenemos empaquetada la última versión y que además tenemos en disco el *tarball* del autor.

Una vez que disponemos de la nueva versión del programa, debemos crear una nueva entrada en el fichero `debian/changelog` (que se puede automatizar en parte con el programa `dch`). Para aplicar los cambios del nuevo tarball puedes utilizar el programa `uupdate` aunque `uscan` puede encargarse también de esto. A continuación debe comprobar que la construcción de la nueva versión es correcta poniendo especial atención a la aplicación de los parches sobre la nueva versión del código fuente.

[6] Como ejemplo vea `http://qa.debian.org/developer.php?login=pkg-games-devel%40lists.alioth.debian.ora`

25.2.7. Subir un paquete a Debian

Enviar el paquete resultante a un repositorio oficial de Debian requiere el uso de algunas herramientas adicionales (la principal es `dupload` que consideramos exceden el ámbito de este documento. Si estás interesado en que tus paquetes aparezcan en los repositorios oficiales lo más sencillo es conseguir un *sponsor* y él te ayudará con esa tarea, auditará tu trabajo de empaquetado y subirá el paquete por ti.

25.3. Otros formatos de paquete

Junto con `.deb`, el otro sistema de paquete de amplio uso es RPM (RPM Package Manager). Este formato fue creado por la distribución Red Hat y hoy en día es utilizado por muchas otras: Fedora, SUSE, CentOS, Yellow Dog, Oracle Linux, etc.

El formato `.rpm` guarda muchas similitudes con `.rpm`. Existen paquetes de fuentes (con extensión `.srpm` o `src.rpm`), paquetes binarios por cada arquitectura (por ejemplo `.i386.rpm`) y también independientes de arquitectura (`.noarch.rpm`). El procedimiento para la creación de un paquete `.rpm` es en muchos sentidos más anárquico que el equivalente `.deb` e históricamente los gestores de paquetes manejas peor las dependencias (en especial las circulares).

Sin embargo, es sencillo crear un paquete `.rpm` a partir del equivalente `.deb` mediante el programa `alien`:

```
$ fakeroot alien --to-rpm ../ogre-hello_0.1-1_amd64.deb
ogre-hello-0.1-2.x86_64.rpm generated
```

Capítulo 26

Representación Avanzada

Sergio Pérez Camacho
Jorge López González
César Mora Castro

Los sistemas de partículas suponen una parte muy importante del impacto visual que produce una aplicación 3D. Sin ellos, las escenas virtuales se compondrían básicamente de geometría sólida y texturas. Los elementos que se modelan utilizando estos tipos de sistemas no suelen tener una forma definida, como fuego, humo, nubes o incluso aureolas simulando ser escudos de fuerza. En las siguientes secciones se van a introducir los fundamentos de los sistemas de partículas y billboards, y cómo se pueden generar utilizando las múltiples características que proporciona Ogre.

Figura 26.1: Ejemplo típico de videojuego que utiliza billboards para modelar la vegetación. Screenshot tomado del videojuego libre Stunt Rally.

26.1. Fundamentos

Para poder comprender cómo funcionan los sistemas de partículas, es necesario conocer primero el concepto de Billboard, ya que dependen directamente de estos. A continuación se define qué son los Billboard, sus tipos y los conceptos matemáticos básicos asociados a estos, para despues continuar con los sistemas de partículas.

26.1.1. Billboards

Billboard significa, literalmente, *valla publicitaria*, haciendo alusión a los grandes carteles que se colocan cerca de las carreteras para anunciar un producto o servicio. Aquellos juegos en los que aparece el nombre de un jugador encima de su personaje, siempre visible a la cámara, utilizan billboards. Los árboles de juegos de carrera que daban la sensación de ser un dibujo plano, utilizan Billboards.

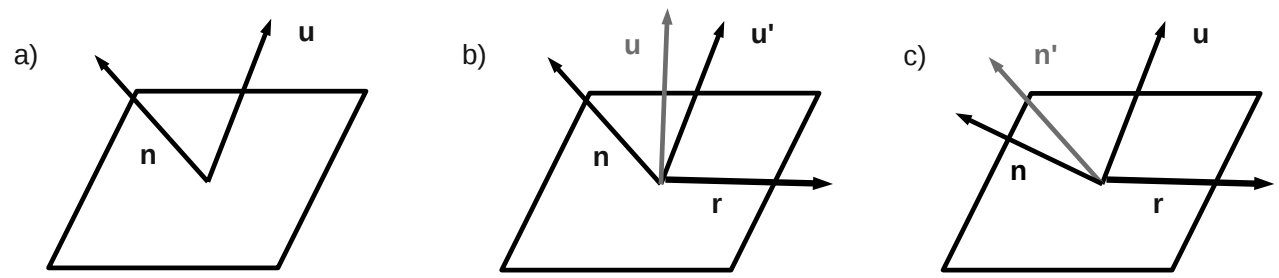


Figura 26.2: En a) se puede observar un Billboard con el vector up y el vector normal. En b) y c) se muestran ejemplos de cómo calcular uno de los vectores dependiendo del otro en el caso de no ser perpendiculares.

Un Billboard es un polígono con una textura, y con un vector de orientación. A medida que la posición de la cámara cambia, la orientación del Billboard cambiará.

Esta técnica se combina con transparencia mediante un canal alfa y con texturas animadas para representar vegetación (especialmente hierba), humo, niebla, explosiones...

Cada billboard está formado por un vector normal $\vec{n}$ y un vector up $\vec{u}$, como se aprecia en la Figura 26.2. Esto es suficiente para definir la orientación del polígono, y poder extraer así la matriz de rotación. El último dato necesario es la *posición* del polígono en el espacio.

Para que el billboard sea correcto, el vector normal y el up deben ser perpendiculares. A menudo no lo son, por lo que hay que utilizar una serie de transformaciones para conseguirlo. El método consiste en tomar uno de los dos vectores como *fijo*, mientras que será el otro el que se recalcule.

Una vez tomado un vector fijo, se calcula el vector $\vec{r}$, resultado de realizar el producto vectorial entre $\vec{u}$ y $\vec{n}$, por lo que será perpendicular a ellos:

$$\vec{r} = \vec{u} \cdot \vec{v}$$

El siguiente paso es normalizarlo, pues se tomará como vector canónico para la matriz de rotación del billboard.

En caso de haber tomado como vector fijo el vector normal $\vec{n}$ (como representa el caso b) de la Figura 26.2), se calcula el nuevo vector $\vec{u'}$ mediante:

$$\vec{u'} = \vec{n} \cdot \vec{r}$$

En el caso de haber sido el vector up $\vec{u}$ el escogido como fijo (caso c) de la Figura 26.2), la ecuación es la que sigue:

$$\vec{n'} = \vec{r} \cdot \vec{u}$$

El nuevo vector es normalizado, y ya se podría obtener la matriz de rotación del billboard. El criterio para escoger un vector como fijo para calcular la orientación del billboard depende del tipo de este, y del efecto que se quiera obtener. A continuación se explican los tres tipos básicos de billboard existentes.

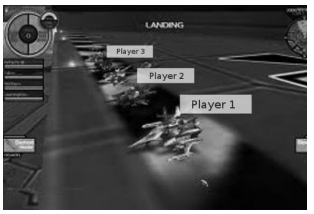


Figura 26.3: *Screenshot* de billboard alineado con la pantalla. Obtenido del videojuego libre Air Rivals.

Billboard alineado a la pantalla

Estos son los tipos más simples de billboard. Su vector up $\vec{u}$ siempre coincide con el de la cámara, mientras que el vector normal $\vec{n}$ se toma como el inverso del vector normal de la cámara (hacia donde la cámara está mirando). Estos dos vectores son siempre perpendiculares, por lo que no es necesario recalcular ninguno con el método anterior. La matriz de rotación de estos billboard es la misma para todos.

Por lo tanto estos billboard siempre estarán alienados con la pantalla, aun cuando la cámara realice giros sobre el eje Z (*roll*), como se puede apreciar en la Figura 26.3. Esta técnica también puede utilizarse para *sprites* circulares como partículas.

Billboard orientado en el espacio

En el caso de tratarse de un objeto físico, en el que el vector up debe corresponder con el vector up del mundo, el tipo anterior no es el apropiado. En este caso, el vector up del billboard no es de la cámara, si no el del mundo virtual, mientras que el vector normal sigue siento la inversa del vector hacia donde mira la cámara. En este caso el vector fijo es el normal, mientras que el que se recalcula es el vector up.

Sin embargo, utilizar esta misma matriz de rotación para todos los billboard puede dar lugar a imprecisiones. Según se ha explicado, estos tipos de billboard se mostrarían como en el caso a) de la Figura 26.5. Al estar lo suficientemente cerca de la cámara puede sufrir una distorsión debido a la perspectiva del punto de vista.

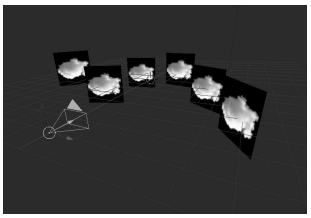


Figura 26.4: Ejemplo de utilización de billboards orientados en el espacio.

La solución a esta distorsión son los billboard orientados al punto de vista. El vector up seguiría siendo el mismo (el vector up del mundo), mientras que el vector normal es el que une el centro del billboard con la posición de la cámara. De esta forma, cada billboard tendría su propia matriz de rotación, y dicha distorsión no existiría. En el caso b) de la Figura 26.5 se aprecia este último tipo.

Evidentemente, este tipo de billboard es menos eficiente, ya que cada uno debe calcular su propia matriz de rotación. Para paliar este aumento de consumo de tiempo de cómputo, podría implementarse dos niveles de billboard dependiendo de la distancia de estos a la cámara. Si están lo suficientemente lejos, la distorsión es mínima y pueden utilizarse los primeros, mientras que a partir de cierta distancia se aplicarían los billboard orientados al punto de vista.

Los billboard orientados en el espacio son muy útiles para la representación de llamas, humo, explosiones o nubes. Una técnica que se suele utilizar es añadir una textura animada a un billboard, y luego crear de forma caótica y aleatoria instancias de este billboard, cambiando parámetros como el tamaño o la transparencia. De esta forma se elimina la sensación de bucle repetitivo en la animación. Este es un método muy común para representar algunos sistemas de partículas. En [101] se describe la técnica utilizada para la implementación de las nubes en Microsoft Flight Simulator.

Un inconveniente de esta forma de implementar sistemas de partículas es cuando intersectan con objetos con geometría real. Al realizarse esta intersección se pierde la ilusión. Para ello se implementa un *fade-out*, que consiste en hacer que el billboard sea más transparente a medida que se acerca a cualquier objeto. Este tipo de billboards se denominan *soft particles*.

En [7] se pueden encontrar más técnicas utilizadas para dar más sensación de realismo a estos billboard.

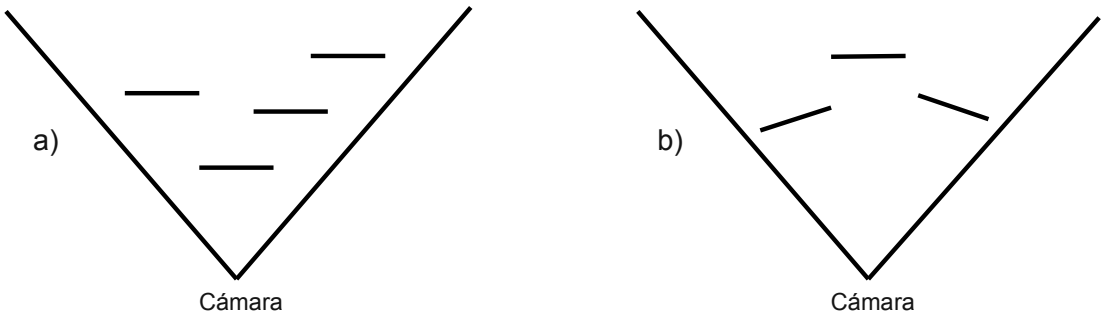


Figura 26.5: Billboards con el mismo vector normal que la cámara en a), mientras que en b) son orientados al punto de vista.

Billboard axial

El último tipo de billboard son los axiales. Estos no miran directamente a la cámara, simplemente giran alrededor de un eje fijo definido, normalmente su vector up. Este técnica suele utilizarse para representar árboles lejanos, como el de la Figura 26.1. En este caso el vector fijo es el vector up, mientras que el recalculado es el normal.

El mayor problema con este tipo de billboard es que si la cámara se sitúa justo encima del billboard (en algún punto de su eje de rotación), la ilusión desaparecía al no mostrarse el billboard. Una posible solución es añadir otro billboard horizontal al eje de rotación. Otra sería utilizar el billboard cuando el modelo esté lo suficientemente lejos, y cambiar a geometría tridimensional cuando se acerque.

26.1.2. Sistemas de partículas

Como se ha visto en la sección anterior, con los billboard se pueden representar de forma eficiente y visualmente efectiva muchos tipos de elementos como nubes, humo o llamas. A continuación se va a explicar de forma más concreta en qué consiste un sistema de partículas y qué técnicas se utilizan para implementarlos.

Según [73], un sistema de partículas es un conjunto de pequeños objetos separados en movimiento de acuerdo a algún algoritmo. Su objetivo principal es la simulación de fuego, humo, explosiones, flujos de agua, árboles, etc.

Las tareas típicas de un sistema de partículas incluyen la creación, puesta en movimiento, transformación y eliminado de dichas partículas durante sus diferentes periodos de vida. Sin embargo, la que más nos interesa es la representación de dichas partículas.

Figura 26.6: Sistema de partículas simulando una llamarada.

Dependiendo del elemento que se quiera representar, las partículas pueden ser mostradas como simples puntos, líneas o incluso billboards. Algunas bibliotecas gráficas como DirectX da soporte para la representación de puntos, y eliminar así la necesidad de crear un billboard con un polígono.

Algunos sistemas de partículas pueden implementarse mediante el *vertex shader*, para calcular la posición de las distintas partículas y así delegar esa parte de cómputo a la GPU. Además puede realizar otras tareas como detección de colisiones.

Elementos de vegetación como hierba o árboles pueden realizarse mediante estas partículas, y determinar la cantidad de estas dependiendo de la distancia de la cámara, todo mediante el *geometry shader*.

Figura 26.8: Modelado de un helicóptero mediante nube de billboards.

A continuación se explican con más detalle dos conceptos básicos utilizados en los sistemas de partículas: los *impostors* y las *nubes de billboards*.

Impostors

Un *impostor* es un billboard cuya textura es renderizada en tiempo de ejecución a partir de la geometría de un objeto más complejo desde la posición actual de la cámara. El proceso de rendering es proporcional al número de píxeles que el impostor ocupa en la pantalla, por lo que es mucho más eficiente. Un buen uso para estos impostors es para representar un elemento que esté compuesto por muchos objetos pequeños iguales, o para objetos muy lejanos. Además, dependiendo de la distancia, la frecuencia con la que se renderizan esos objetos es menor.

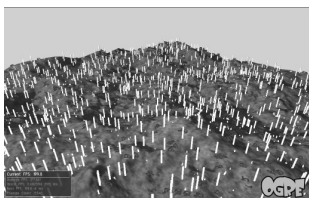


Figura 26.7: Ejemplo de utilización de impostors utilizando Ogre.

Una ventaja importante de los impostors es el poder añadir un desenfoque a la textura de forma rápida para poder dar la sensación de profundidad de campo.

Nubes de Billboards

El problema de los impostors es que estos deben continuar orientadas a la cámara, por lo que si esta hace un movimiento el impostor debe ser renderizado de nuevo. Las nubes de billboards consiste en representar un modelo complejo como un conjunto pequeño de billboards superpuestos. A cada uno de ellos se les puede aplicar un material y propiedades diferentes, para conseguir un buen resultado con un alto rendimiento. En la Figura 26.8 se puede apreciar el modelo geométrico de un helicóptero (izquierda), su correspondiente representación con una nube de billboards (centro) y la descomposición de cada billboard (derecha). Este ejemplo ha sido tomado de [26].

Algunas aproximaciones propuestas sugieren utilizar parte del modelo como geometría convencional, y para el resto utilizar una nube de billboards. Por ejemplo, hacer el tronco de un arbol geométricamente y añadir billboards para las hojas.

26.2. Uso de Billboards

Billboards individuales

También es posible controlar la representación individual de cada *Billboard* dentro de un *BillboardSet*, pero la penalización en rendimiento es mucho mayor. En la mayoría de los casos es más eficiente crear distintos *BillboardSets*.

En Ogre, no existe un elemento billboard por sí sólo que se pueda representar. Estos deben pertenecer a un objeto de la clase *BillboardSet*. Esta clase se encarga de *gestionar* los diferentes billboards que están contenidos en ella.

Todos los *Billboard* que pertenezcan a un *BillboardSet* deben ser idénticos en cuanto a tamaño y material. Este es un requerimiento semi-obligatorio por cuestiones de eficiencia. Una vez añadidos, es posible cambiar los *Billboard* individualmente, aunque esto se desaconseja debido a la penalización en el rendimiento, a no ser que haya una buena razón (por ejemplo, volúmenes de humo que se expanden).

Ogre tratará el *BillboardSet* como un elemento único: o se representan todos los *Billboard* contenidos, o no se representa ninguno. El posicionamiento de los *Billboards* se realiza relativo a la posición del *SceneNode* al que pertenece el *BillboardSet*.

La forma más común de crear un *BillboardSet* es indicando el número de *Billboard* que son necesarios en el constructor del *BillboardSet*. Cada vez que queramos crear un *Billboard* mediante el método *createBillboard* de *BillboardSet*, se nos devolverá uno. Una vez que se agote la capacidad el método devolverá *NULL*.

Capacidad dinámica

Se puede indicar al *BillboardSet* que aumente de forma dinámica su capacidad. Así, cada vez que se pida un Billboard y no quede ninguno, se aumentará la capacidad al doble automáticamente. Este método es potencialmente peligroso, sobretodo si se utiliza en algún tipo de bucle.

En el siguiente código vemos un ejemplo sencillo de una escena con tres billboards. Como se ha explicado anteriormente, para poder crear *Billboards* es necesario que estos pertenezcan a un *BillboardSet*. En la línea [2] creamos uno, con nombre *BillboardSet* y con capacidad para tres *Billboard*. Por defecto, el tipo de *Billboard* es *point*. Más adelante se explicarán los tres tipos básicos que ofrece Ogre. En la línea [3] se asigna un material a esos Billboard. En este caso, el material se encuentra descrito en un fichero llamado *Cloud.material*. En la línea [4] se especifica el tamaño del rectángulo que definirán cada uno de los *Billboard*. En la línea [5] se activa la opción de que Ogre ordene automáticamente los *Billboard* según su distancia a la cámara, para que, en caso de que el material tenga transparencia, no se superpongan unos encima de otros y causen un efecto indeseado.

Listado 26.1: Primer ejemplo con Billboards

```
1  void MyApp::createScene() {
2    Ogre::BillboardSet* billboardSet = _sceneManager->
         createBillboardSet("BillboardSet",3);
3    billboardSet->setMaterialName("Cloud");
4    billboardSet->setDefaultDimensions(10.,10.);
5    billboardSet->setSortingEnabled(true);
6
7    billboardSet->createBillboard(Ogre::Vector3(0,0,0));
8    billboardSet->createBillboard(Ogre::Vector3(50,0,50));
9    billboardSet->createBillboard(Ogre::Vector3(-50,0,-50));
10
11   Ogre::SceneNode* node1 = _sceneManager->createSceneNode("Node1");
12   node1->attachObject(billboardSet);
13   _sceneManager->getRootSceneNode()->addChild(node1);
14 }
```

De las líneas [7-9] se crean tantos *Billboard* como capacidad se dió al *BillboardSet* en el momento de su creación. A cada uno de ellos se le indica la posición relativa al *SceneNode* al que pertenece el *BillboardSet*. Por último, en las líneas [11-13] se crea un nodo y se adjunta el *BillboardSet* a él.

26.2.1. Tipos de Billboard

Ogre, por defecto, al crear un *Billboard* lo crea utilizando el tipo *point billboard*. Este tipo se puede cambiar a través del método *BillboardSet::setBillboardType*, y recibe un argumento del tipo *BillboardType*. Los tres tipos básicos existentes son:

- **Point Billboard:** se indican con el valor *BBT_POINT*, y se corresponden con los Billboards alineados en la pantalla. Se trata del tipo más simple, y no es necesario indicar ningún parámetro adicional.
- **Oriented Billboard:** se indican con los valores:
  - *BBT_ORIENTED_COMMON*
  - *BBT_ORIENTED_SELF*

  Se corresponden con los Billboard axiales. Es necesario indicar un eje sobre el cuál girar (en el ejemplo de los árboles, se corresponde con el tronco). En caso de utilizarse la opción *Oriented Common*, este vector se indica mediante el método *BillboardSet::setCommonDirection*, y todos los *Billboard* del conjunto lo utilizarán. En el caso de utilizar *Oriented Self*, cada *Billboard* podrá tener su propio "tronco", y se especifica en la variable pública *Billboard::mDirection* de cada *Billboard*.
- **Perpendicular Billboard:** se indican con los valores:
  - *BBT_PERPENDICULAR_COMMON*
  - *BBT_PERPENDICULAR_SELF*

  Se corresponde con los Billboard orientados en el espacio. Siempre apuntan a la cámara, pero al contrario que con los Billboard alineados en la pantalla, el vector up no tiene por qué ser el mismo que el de ella. En cualquier caso es necesario indicar un vector up mediante la llamada *Billboard::setCommonUpVector*. También se debe indicar un vector de dirección. En caso de haber escogido *Perpendicular Common* se indica mediante la llamada *BillboardSet::setCommonDirection*. En caso de haber escogido *Perpendicular Self*, se almacena en la variable pública *Billboard::mDirection*. Este vector se escoge como fijo, y se recalcula el vector up, según el método explicado en la primera sección. Este vector suele ser el inverso al vector de dirección de la cámara, o el vector con origen en la posición del *Billboard* y destino la posición de la cámara. Es importante normalizarlo.

En el caso de los tipos *BBT_PERPENDICULAR_COMMON*, *BBT_PERPENDICULAR_SELF* y *BBT_ORIENTED_SELF* es necesario actualizar los valores de los vectores según la posición actual de la cámara en cada frame. Para ello se debe recuperar los *Billboard* desde el método *frameStarted* del *FrameListener*, y actualizarlos según el valor escogido.

26.2.2. Aplicando texturas

Hasta ahora se ha explicado que todos los *Billboard* de un mismo *BillboardSet* deben tener el mismo tamaño y el mismo material. Sin embargo, exite la opción de indicar a cada *Billboard* qué porción de la textura del material puede representar. De esta forma, se puede indicar una textura que esté dividida en filas y en columnas con varias *subtexturas*, y asignar a cada *Billboard* una de ellas.

En la Figura 26.9 se observa un ejemplo de textura subdividida. Las coordenadas están contenidas en el rango de 0 a 1.

En el siguiente código se va a utilizar esta técnica para que, cada uno de los *Billboard*, muestren un trozo de la textura asociada. En 2-9 se crea el escenario y un *BillboardSet* de tipo *point* y un material asociado llamado "ABC", como la de la Figura 26.9. De las líneas 11-19 se instancian los 4 *Billboard*. Se ha declarado el puntero a *Billboard b* para poder referenciar cada uno de los *Billboard* según se crean y poder indicar las coordenadas de las texturas asociadas. Esto se hace mediante el método *Billboard::setTexcoordRect*, al cual se le pasa un objeto del tipo *FloatRec*, indicando las coordenadas de la esquina superior izquierda y de la esquina inferior derecha. Los valores de esa esquina están en el rango de 0 a 1. Para terminar, de las líneas 21-23 se crea el nodo y se adjunta el *BillboardSet*.

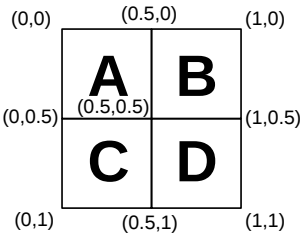


Figura 26.9: Ejemplo de subdivisión de una textura.

Listado 26.2: Ejemplo de coordenadas de texturas.

```
 1 void MyApp::createScene() {
 2   /* Preparing simbad and ground */
 3   /* Creating Sinbad and Ground ...*/
 4
 5   /* ABC BillboardSet*/
 6   Ogre::BillboardSet* abcBillboardSet = _sceneManager->
         createBillboardSet("AbcBillboardSet",4);
 7   abcBillboardSet->setBillboardType(Ogre::BBT_POINT);
 8   abcBillboardSet->setMaterialName("ABC");
 9   abcBillboardSet->setDefaultDimensions(7.,7);
10
11   Ogre::Billboard* b;
12   b = abcBillboardSet->createBillboard(Ogre::Vector3(-15,0,0));
13   b->setTexcoordRect(Ogre::FloatRect(0.,0.,0.5,0.5));
14   b = abcBillboardSet->createBillboard(Ogre::Vector3(-5,0,0));
15   b->setTexcoordRect(Ogre::FloatRect(0.5,0.,1.,0.5));
16   b = abcBillboardSet->createBillboard(Ogre::Vector3(5,0,0));
17   b->setTexcoordRect(Ogre::FloatRect(0.,0.5,0.5,1.));
18   b = abcBillboardSet->createBillboard(Ogre::Vector3(15,0,0));
19   b->setTexcoordRect(Ogre::FloatRect(0.5,0.5,1.,1.));
20
21   Ogre::SceneNode* abcNameNode =_sceneManager->getRootSceneNode()->
         createChildSceneNode("AbcNameNode");
22   abcNameNode->setPosition(0,10,0);
23   abcNameNode->attachObject(abcBillboardSet);
24 }
```

26.3. Uso de Sistemas de Partículas

Los sistemas de partículas en Ogre se implementan típicamente mediante scripts, aunque cualquier funcionalidad se puede realizar también por código. La extensión de los script que definen las plantillas de estos sistemas es *.particle*. Son plantillas porque en ellas se definen sus características, y luego se pueden instanciar tanto sistemas como se desee. Es decir, se puede crear un fichero *.particle* para definir un tipo de explosión, y luego instanciar cualquier número de ellas.

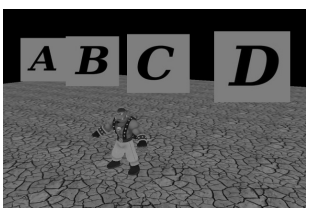


Figura 26.10: Resultado del ejemplo de Billboard con coordenadas de texturas.

Los sistemas de partículas son entidades que se enlazan a *SceneNodes*, por lo que están sujetos a la orientación y posicionamiento de estos. Una vez que se han emitido las partículas, estas pasan a formar parte de la escena, por lo que si se mueve el punto de emisión del sistema, las partículas no se verán afectadas, quedando en el

mismo sitio. Esto es interesante si se quiere dejar una estela, por ejemplo, de humo. Si se desea que las partículas ya creadas se trasladen con el nodo al que pertenece el sistema, se puede indicar que el posicionamiento se haga referente al sistema de coordenadas local.

Los sistemas de partículas deben tener siempre una cantidad límite de estas, o *quota*. Una vez alcanzado esta cantidad, el sistema dejará de emitir hasta que se eliminen algunas de las partículas *antiguas*. Las partículas tienen un límite de vida configurable para ser eliminadas. Por defecto, este valor de *quota* es 10, por lo que puede interesar al usuario indicar un valor mayor en la plantilla.

Eficiencia ante todo

Los sistemas de partículas pueden rápidamente convertirse en una parte muy agresiva que requiere mucho tiempo de cómputo. Es importante dedicar el tiempo suficiente a optimizarlos, por el bien del rendimiento de la aplicación.

Ogre necesita calcular el espacio físico que ocupa un sistema de partículas (su *BoundingBox*) de forma regular. Esto es computacionalmente costoso, por lo que por defecto, deja de recalcularlo pasados 10 segundos. Este comportamiento se puede configurar mediante el método *ParticleSystem::setBoundsAutoUpdated()*, el cual recibe como parámetro los segundos que debe recalcular la *BoundingBox*. Si se conoce de antemano el tamaño aproximado del espacio que ocupa un sistema de partículas, se puede indicar a Ogre que no realice este cálculo, y se le indica el tamaño fijo mediante el método *ParticleSystem::setBounds()*. De esta forma se ahorra mucho tiempo de procesamiento. Se puede alcanzar un compromiso indicando un tamaño inicial aproximado, y luego dejando a Ogre que lo recalcule durante poco tiempo pasados algunos segundos desde la creación del sistema.

A continuación se describen los dos elementos básicos que definen los sistemas de partículas en Ogre: los *emisores* y los *efectores*.

26.3.1. Emisores

Los emisores definen los *objetos* que literalmente *emiten* las partículas a la escena. Los distintos emisores que proporciona Ogre son:

- **Puntos:** *point*. Todas las partículas son emitidas desde un mismo punto.
- **Caja:** *box*. Las partículas son emitidas desde cualquier punto dentro de un volumen rectangular.
- **Cilindro:** *cylinder*. Las partículas son emitidas desde un volumen cilíndrico definido.
- **Elipsoide:** *ellipsoid*. Las partículas se emiten desde un volumen elipsoidal.
- **Superficie de elipsoide:** *hollow elipsoid*. Las partículas se emiten desde la superficie de un volumen elipsoidal.
- **Anillo:** *ring*. Las partículas son emitidas desde los bordes de un anillo.

La velocidad, frecuencia y dirección de emisión de las partículas es completamente configurable. Estos emisores se posicionan de forma relativa al *SceneNode* al que pertenecen.

Las partículas no son emitidas en una línea recta. Se debe especificar un ángulo mediante el parámetro *angle* para definir el *cono* de emisión. Un valor 0 indica que se emiten en línea recta, mientras que un valor de 180 significa que se emite en cualquier dirección. Un valor de 90 implica que se emiten de forma aleatoria en el hemisferio centrado en el vector de dirección.

Otros parámetros que se pueden configurar son la frecuencia de emisión (partículas/segundo), la velocidad (puede ser una velocidad establecida o aleatoria para cada partícula), el tiempo de vida o TTL (Time To Live) (definido o aleatorio), y el tiempo de emisión del sistema. Más adelante se mostrarán ejemplos de uso de estos parámetros.

26.3.2. Efectores

Los efectores o *affectors* realizan cambios sobre los sistemas de partículas. Estos cambios pueden ser en su dirección, tiempo de vida, color, etc. A continuación se explican cada uno de los efectores que ofrece Ogre.

- **LinearForce:** aplica una *fuerza* a las partículas del sistema. Esta fuerza se indica mediante un vector, cuya dirección equivale a la dirección de la fuerza, y su módulo equivale a la magnitud de la fuerza. La aplicación de una fuerza puede resultar en un incremento enorme de la velocidad, por lo que se dispone de un parámetro, *force_application* para controlar esto. El valor *force_application average* ajusta el valor de la fuerza para estabilizar la velocidad de las partículas a la media entre la magnitud de la fuerza y la velocidad actual de estas. Por el contrario, *force_application add* deja que la velocidad aumente o se reduzca sin control.
- **ColourFader:** modifica el color de una partícula mientras ésta exista. El valor suministrado a este modificador significa ‘ ‘ cantidad de cambio de una componente de color en función del tiempo”. Por lo tanto, un valor de *red -0.5* decrementará la componente del color rojo en 0.5 cada segundo.
- **ColourFader2:** es similar a *ColourFader*, excepto que el modificador puede cambiar de comportamiento pasada una determinada cantidad de tiempo. Por ejemplo, el color de una partícula puede decrementarse suavemente hasta el 50 % de su valor, y luego caer en picado hasta 0.
- **ColourInterpolator:** es similar a *ColourFader2*, sólo que se pueden especificar hasta 6 cambios de comportamiento distintos. Se puede ver como una generalización de los otros dos modificadores.
- **Scaler:** este modificador cambia de forma proporcional el tamaño de la partícula en función del tiempo.
- **Rotator:** rota la textura de la partícula por bien un ángulo aleatorio, o a una velocidad aleatoria. Estos dos parámetros son definidos dentro de un rango (por defecto 0).
- **ColourImage:** este modificador cambia el color de una partícula, pero estos valores se toman de un fichero imagen (con extensión .png, .jpg, etc.). Los valores de los píxeles se leen de arriba a abajo y de izquierda a derecha. Por lo tanto, el valor de la esquina de arriba a la izquierda será el color inicial, y el de abajo a la derecha el final.

ColourFader

Un valor de -0.5 no quiere decir que se reduzca a la mitad cada segundo, y por lo tanto, nunca alcanzará el valor de 0. Significa de al valor de la componente (perteneciente al intervalo de 0 a 1) se le restará 0.5. Por lo tanto a un valor de blanco (1) se reducirá a negro (0) en dos segundos.

26.3.3. Ejemplos de Sistemas de Partículas

Para poder utilizar los sistemas de partículas en Ogre, es necesario editar el fichero *plugins.cfg* y añadir la línea *Plugin=Plugin_ParticleFX* para que pueda encontrar el plugin. En caso de utilizarse en Windows, hay que asegurarse de tener la biblioteca *Plugin_ParticleFX.dll* en el directorio de plugins del proyecto, o la biblioteca *Plugin_ParticleFX.so* en caso de sistemas UNIX

Este primer ejemplo ilustra un anillo de fuego. A continuación se explica paso a paso cómo se instancia el sistema de partículas y qué significan cada uno de los campos del script que lo define.

Listado 26.3: Instancia de un sistema de partículas.

```
1 void MyApp::createScene() {
2   Ogre::ParticleSystem* ps = _sceneManager->createParticleSystem("
        Ps","ringOfFire");
3
4   Ogre::SceneNode* psNode = _sceneManager->getRootSceneNode()->
        createChildSceneNode("PsNode");
5   psNode->attachObject(ps);
6 }
```

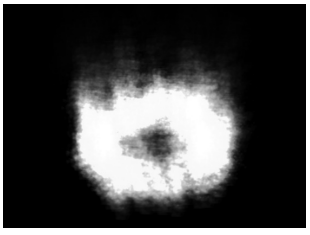


Figura 26.11: Captura de pantalla del sistema de partículas *RingOfFire*

Como se puede observar, crear un sistema de partículas en código no es nada complicado. En la línea [2] se crea un objeto de tipo *ParticleSystem*, indicando su nombre y el nombre del script que lo define, en este caso *ringOfFire*. En las líneas [4-5] se crea un *SceneNode* y se añade a él.

El siguiente es el script que define realmente las propiedades del sistema de partículas.

Los scripts de sistemas de partículas comienzan con la palabra reservada *particle_system* seguido del nombre del sistema, como se indica en la línea [1]. De las líneas [3-6] se indican varios parámetros generales del sistema:

- *quota:* Indica el número máximo de partículas que pueden haber vivas en un momento. Si el número de partículas alcanza esta cantidad, no se crearán más partículas hasta que mueran otras.
- *material:* indica el material de cada partícula. Cada una de las partículas es, por defecto, un *Billboard* como los estudiados anteriormente. Este material indica qué textura se representará en cada uno de ellos.
- *particle_width:* indica el ancho de cada partícula.
- *particle_height:* indica el alto de cara partícula.

A continuación se declaran tantos emisores y modificadores como se deseen. En las líneas [8-19] se declara un emisor del tipo anillo. Los parámetros especificados para este emisor son:

- *angle:* define el ángulo de apertura con el que las partículas salen disparadas.
- *direction:* indica la dirección de salida d e las partículas, teniendo en cuenta el ángulo. En realidad, el par dirección-ángulo define un cono por el cual las partículas se crean.

¡Y aún hay más!

Existen multitud de parámetros para configurar los sistemas de partículas, emisores y modificadores en Ogre. En la API oficial se detallan todos y cada uno de estos parámetros.

Listado 26.4: Script para el sistema *ringOfFire*

```
particle_system ringOfFire
{
    quota   1000
    material    explosion
    particle_width  10
    particle_height 10

    emitter Ring
    {
        angle   10
        direction   0 1 0
        emission_rate   250
        velocity_min    3
        velocity_max    11
        time_to_live    3
        width   30
        height  30
        depth   2
    }

    affector ColourFader
    {
        red -0.5
        green   -0.5
        blue    -0.25
    }
}
```

- *emission_rate:* indica el ratio de partículas por segundo emitidas.
- *velocity_min:* velocidad mínima inicial de las partíclas.
- *velocity_max:* velocidad máxima inicial de las partículas.
- *time_to_live* tiempo de vida de una partícula.
- *width:* determina el ancho del anillo.
- *height:* determina el alto del anillo.
- *depth:* determina la profundidad del anillo.

Además se ha declarado un modificado del tipo *ColourFader* que cambia el color de las partículas en el tiempo. Concretamente, por cada segundo que pasa decrementa la componente del color rojo en 0.5, la del color verde en 0.5 y la del color azul en 0.25.

Año	Tarjeta Gráfica	Hito
1987	IBM VGA	Provee un *pixel framebuffer* que la CPU debe encargarse de llenar.
1996	3dfx Voodoo	Rasteriza y texturiza triángulos con vértices pre-transformados.
1999	NVIDIA GeForce 256	Aplica tanto transformaciones, como iluminación a los vértices. Usa una *fixed function pipeline*.
2001	NVIDIA GeForce 3	Incluye *pixel shader* configurable y *vertex shader* completamente programable.
2003	NVIDIA GeForce FX	Primera tarjeta con shaders completamente programables.

Cuadro 26.1: Evolución del hardware de gráfico para PC.

26.4. Introducción a los Shaders

A lo largo de las siguiente páginas se introducirá, y se enseñará a manejar, una de las herramientas más poderosas que existen a la hora de sacar partido de nuestro hardware gráfico: los *shaders*. Descubriremos cómo gracias a estos pequeños fragmentos de código nos hacemos con todo el control sobre el dibujado de nuestras escenas, a la vez que se abre ante nosotros la posibilidad de añadir una gran variedad de efectos que, hasta no hace mucho, eran imposibles en aplicaciones gráficas interactivas.

Para comenzar a entender lo que son los *shaders* es conveniente acercarse primero a la problemática que resuelven y, para ello, en la siguiente sección se hará un pequeño repaso a la historia de la generación de gráficos por computador, tanto en su vertiente interactiva como en la no interactiva, puesto que la historia de los *shaders* no es más que la historia de la lucha por conseguir mejorar y controlar a nuestro gusto el proceso de generación de imágenes por ordenador.

26.4.1. Un poco de historia

Nuestra narracción comienza en los años 80, una época que hoy en día a algunos les puede parecer como un tiempo mejor, pero que definitivamente no lo era, sobre todo si querías dedicarte al desarrollo de aplicaciones gráficas interactivas.

En estos años el desarrollo de este tipo de aplicaciónes era realmente complicado y no sólo por la poca capacidad de las máquinas de la época, si no también por la falta de estandarización que existía en los APIs gráficos, lo que provocaba que cada hardware necesitara de su propio software para poder ser usado.

Este contexto hacía evidente que los gráficos 3D tardarían todavía un poco en llegar a ser algo común en los ordenadores personales y aun más si hablamos de moverlos en tiempo real. Es por tanto fácil comprender por qué los mayores avances en este campo estuvieron enfocados sobre todo hacia el renderizado no interactivo. En aquellos tiempos la principal utilidad que tenía la informática gráfica eran la investigación y el desarrollo de *CGI (Computer Generated Imagery)* para anuncios y películas.

Es en este punto donde la historia de los *shaders* comienza, en una pequeña empresa conocida como *LucasFilms* a principios de los años 80. Por esas fechas el estudio decidió contratar programadores gráficos para que, comandados por *Edwin Catmull*, empezaran a informatizar la industria de los efectos especiales. Casi nada.

Aquel grupo de pioneros se embarcó en varios proyectos diferentes, logrando que cada uno de ellos desembocara en cosas muy interesantes. Uno de estos proyectillos acabó siendo el germen de lo que más adelante se conocería como *Pixar Animation Studios*. Y fue en esta compañía donde, durante el desarrollo del API abierto *RISpec*

(RenderMan Interface Specification), se creó el concepto de *shader*. El propósito de *RISpec* era la descripción de escenas 3D para convertirlas en imágenes digitales fotorealistas. En este API se incluyó el *RenderMan Shading Language*, un lenguaje de programación al estilo C que permitía, por primera vez, que la descripción de materiales de las superficies no dependiera sólo de un pequeño conjunto de parámetros, sino que pudiera ser especificada con toda libertad (no olvidemos que tener la posibilidad de especificar cómo se hace o no se hace algo es siempre muy valioso).

RISpec 3.0 (donde se introdujo el concepto de *shader*) fue publicado en Mayo de 1988 y fue diseñado con la vista puesta en el futuro para, de esta manera, poder adaptarse a los avances en la tecnología durante un numero significativo de años. A las películas en que se ha usado nos remitimos para dar fe de que lo consiguieron (`http://www.pixar.com/featurefilms/index.html`).

Hasta aquí el nacimiento del concepto *shader*, una funcionalidad incluida en *RISpec 3.0* para especificar sin cortapisas cómo se dibujan las superficies (materiales en nuestra jerga) de los elementos de una escena 3D. ¿Pero que ocurre con nuestro campo de estudio?

El panorama para los gráficos en tiempo real no era tan prometedor por aquel entonces hasta que surgieron los primeros APIs estándar que abstraían el acceso al hardware gráfico. En 1992 apareció *OpenGL*, en 1995 *DirectX* y, ya lanzados, surge en 1996 la primera tarjeta gráfica, la *3Dfx Voodoo*, que liberaba a la CPU de algunas de las tareas que implicaban la representación de gráficos por ordenador.

Tarjeta gráfica y API estándarizado, estos dos ingredientes combinados ya empiezan a permitir la preparación de un plato como Dios manda de gráficos 3D en tiempo real en ordenadores de escritorio. Si, 1996, este es el momento en que la cosa empieza a ponerse seria y se establecen los actuales *pipelines gráficos* (cuyo funcionamiento ya se trató al principio del Módulo 2, pero que, y hay que insistir, es vital comprender para poder desarrollar cualquier aplicación gráfica interactiva, por ello más adelante se vuelven a tratar).

Aun así, en los 90, tanto los APIs como el hardware ofrecían como único *pipeline* de procesamiento gráfico el *fixed-function pipeline* (FFP). El FFP permite varios tipos de configuración a la hora de establecer cómo se realizará el proceso de renderizado, sin embargo estas posibilidades están predefinidas y, por tanto, limitadas. De este modo, aunque el salto de calidad era evidente durante muchos años el renderizado interactivo estuvo muchísimo más limitado que su versión no interactiva.

Pipeline

En ingeniería del software, un pipeline consiste en una cadena etapas de procesamiento, dispuestas de tal forma que la salida de una de estas etapas es la entrada de la siguiente, facilitando con ello la ejecución en paralelo de las etapas.

La programación gráfica antes de los *shaders* usaba un conjunto fijo de algoritmos que, colectivamente, son conocidos como *fixed-function pipeline*. Este permitía habilitar un conjunto prefijado de características y efectos, pudiendo manipular algunos parámetros pero, como es de esperar, con opciones limitadas no se puede ejercer un gran control sobre lo que ocurre en el proceso de renderizado.

En el lado no interactivo del espectro, las arquitecturas de alto rendimiento de renderizado por software usadas para el CGI de las películas permitía llegar muchísimo más lejos en cuanto a la calidad de las imagenes generadas. *RenderMan* permitía a los artistas y programadores gráficos controlar totalmente el resultado del renderizado mediante el uso de este simple, pero potente, lenguaje de programación.

La evolución de la tecnología para construir los hardware gráficos y su correspondiente incremento en la capacidad de procesamiento acabó permitiendo trasladar la idea de *RenderMan* a los ordenadores domésticos en el año 2001, el cual nos trae a dos protagonistas de excepción: la *NVIDIA GeForce3* que introdujo por primera vez un pipeline gráfico programable (aunque limitado) y *DirectX 8.0* que nos dió el API para aprovechar esas capacidades.

Desde entonces, como suele pasar en la informática, con el paso de los años todo fue a mejor, el hardware y los API gráficos no han hecho sino dar enormes saltos hacia delante tanto en funcionalidad como en rendimiento (un ejemplo de la evolución de las capacidades de las tarjetas gráficas se presenta en la tabla 26.1). Lo cual nos ha llevado al momento actual, en el que los FFP han sido casi sustituidos por los *programmable-function pipelines* y sus *shader* para controlar el procesamiento de los gráficos.

Hasta aquí el repaso a la historia de los *shaders*, que nos ha permitido introducir algunos conceptos básicos que siempre os acompañaran si os animáis a seguir por este campo: *pipeline*, *programmable-function pipeline*, *fixed-function pipeline*. Pero...

26.4.2. ¿Y qué es un Shader?

Una de las definiciones clásicas de *shader* los muestra como: *"piezas de código, que implementan un algoritmo para establecer como responde un punto de una superficie a la iluminación"*. Es decir sirven para establecer el color definitivo de los pixeles que se mostrarán en pantalla.

Gestion-Recursos

En algún momento comprenderéis que hacer videojuegos, al final, no es más que gestionar recursos y balancear calidad de imagen con rapidez de ejecución. Por eso conocer el funcionamiento de un motor gráfico, conocimientos altos de ingeniería del software y gestión de recursos en un programa software son de las habilidades más demandadas.

Como veremos esta definición ya no es del todo correcta. Los *shader* actuales cumplen muchas más funciones, ya que pueden encargarse de modificar todo tipo de parámetros (posición de vértices, color de pixeles e incluso generar geometría en los más avanzados). Hoy por hoy una definición de *shader* más acertada podría ser: *"Un conjunto de software que especifica la forma en que se renderiza un conjunto de geometría"*, definición extraida del libro *"GPU Gems"* [33], ideal para aprender sobre técnicas avanzadas de programación gráfica.

Desde un punto de vista de alto nivel los *shader* nos permiten tratar el estado de renderizado como un recurso, lo cual los convierte en una herramienta extremadamente poderosa, permitiendo que el dibujado de una aplicación, es decir, la configuración del dispositivo que se encarga de ello sea casi completamente dirigido por recursos (de la misma forma en que la geometría y las texturas son recursos externos a la aplicación). Facilitando de esta manera su reusabilidad en múltiples proyectos.

Las definiciones, en mi opinión, suelen ser siempre algo esotéricas para los profanos en una materia, por ello la mejor manera de terminar de explicar qué son y cómo funcionan estas pequeñas piezas de código es repasando el funcionamiento del *pipeline* gráfico: desde la transmisión de los vértices a la *GPU*, hasta la salida por pantalla de la imagen generada.

26.4.3. Pipelines Gráficos

En esta sección se va a repasar el funcionamiento interno de los dos tipos de *pipelines gráficos*, para que de esta forma quede clara la forma en que trabajan los *shader* y cual es exáctamente la función que desempeñan en el proceso de renderizado. Además es muy importante conocer el funcionamiento del *pipeline* a la hora de desarrollar *shaders* y en general para cualquier aplicación con gráficos en tiempo real.

La información ofrecida aquí complementa a la que aparece en las secciones 1, 2 y 3 del capitulo *Fundamentos de Gráficos Tridimensionales* del Módulo 2, donde se explica con mas detalle el *pipeline gráfico* y que se recomienda se repase antes de continuar.

¿Por qué una GPU?

El motivo por el que se usan tarjetas gráficas busca que la CPU delegue en la medida de lo posible la mayor cantidad de trabajo en la GPU, y que así la CPU quede liberada de trabajo. Delegar ciertas tareas en alguien o algo especializado en algo concreto es siempre una buena idea. Existen además dos claves de peso que justifican la decisión de usar un hardware específico para las tareas gráficas. La primera es que las CPU son procesadores de propósito general mientras que la tarea de procesamiento de gráficos tiene características muy específicas que hacen fácil extraer esa funcionalidad aparte. La segunda es que hay mercado para ello, hoy en día nos encontramos con que muchas aplicaciones actuales (videojuegos, simulaciones, diseño gráfico, etc...) requieren de *rendering* interactivo con la mayor calidad posible.

Diseño de una GPU

La tarea de generar imagenes por ordenador suele implicar el proceso de un gran número de elementos, por suerte se da la circunstancia de que la mayor parte de las veces no hay dependencias entre ellos. El procesamiento de un vértice, por ejemplo, no depende del procesamiento de los demás o, tomando un caso en el que quisieramos aplicar iluminación local, se puede apreciar que la representación de un pixel iluminado no depende de la de los demás.

Además el procesamiento de los gráficos es áltamente paralelizable puesto que los elementos involucrados suelen ser magnitudes vectoriales reales. Tanto la posición, como el color y otros elementos geométricos se representan cómodamente mediante vectores a los que se aplican diferentes algoritmos. Estos, generalmente, suelen ser bastante simples y poco costosos computacionalmente (las operaciones más comunes en estos algoritmos son la suma, multiplicación o el producto escalar).

En consecuencia, las GPUs se diseñan como *Stream Processors*. Estos procesadores están especializados en procesar gran cantidad de elementos en paralelo, distribuyéndolos en pequeñas unidades de procesamiento (etapas del *pipeline*), las cuales disponen de operaciones para tratar con vectores de forma eficiente y donde algunas son programables o, al menos, configurables (¿Os suena?).

26.4.4. Fixed-Function Pipeline

Para seguir el proceso tomaremos como referencia el *pipeline gráfico* simplificado de la figura 26.12, que representa las etapas básicas de un *fixed-function pipeline*. Este tipo de *pipeline* no ofrece mucha libertad en la manipulación de sus parámetros pero aun se usa, sobre todo cuando uno quiere rapidez sobre calidad, y sirve como base para entender el por qué son necesarias las etapas programables mediante *shaders*.

Para empezar la aplicación descompone la escena en batches y se los manda al *pipeline*. Así que como entrada, la primera etapa del *pipeline*, recibe el conjunto de vértices correspondientes a la geometría de la escena que va en cada batch.

Primitiva

Una primitiva es una región de una superficie definida por sus vértices. La primitiva más usada en informática gráfica es el triángulo, puesto que sus tres vértices siempre definen un plano.

Un vértice en el contexto en el que nos encontramos se corresponde con la posición de un punto en el espacio 3D, perteneciente a una superficie, para el que se conocen los valores exactos de ciertas propiedades (conocidos como componentes del vértice). Estas propiedades pueden incluir atributos como el color de la superficie (primario y

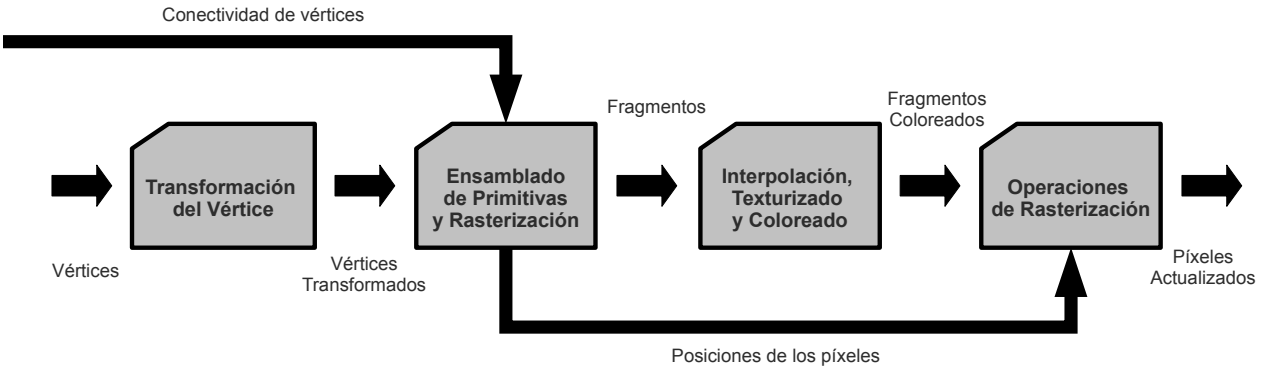


Figura 26.12: Pipeline del hardware gráfico simplificado.

secundario, si tiene componente especular), uno o varios conjuntos de coordenadas de textura, o su vector normal que indicará la dirección en que la superficie está orientada con respecto al vértice y que se usa para los cálculos de iluminación, entre otros (un vértice puede incluir mucha más información, como se muestra en la sección 10.1.2.2 Vertex Atributes del muy recomendable libro Game Engine Architecture [42]).

Batch

Sería conveniente no olvidar jamás que la GPU no suele recibir toda la información de la escena de golpe, sino que se le envían para dibujar varios grupos de primitivas cada vez, agrupados en la unidad de trabajo conocida como *batch*.

Transformación de Vértices

El conjunto de vértices primero atraviesa esta etapa de procesamiento del pipeline gráfico, en la que se realizan una serie de operaciones matemáticas sobre los mismos. Estas operaciones incluyen las transformaciones necesarias para convertir la posición del vértice a la posición que tendrá en pantalla (Transformación de Modelado, Transformación de Visualización y Transformación de Proyección visto en sección 1.1.2 del Módulo 2) y que será usada por el rasterizador, la generación de las coordenadas de textura y el cálculo de la iluminación sobre el vértice para conocer su color.

Ensamblado de Primitivas y Rasterización

Los vértices transformados fluyen en secuencia hacia la siguiente etapa donde el ensamblado de primitivas toma los vértices y los une para formar las primitivas correspondientes gracias a la información recibida sobre la conectividad de los mismos (que indica cómo se ensamblan).

Textura

Es una matriz que almacena los valores de color de una superficie en los puntos interiores de las primitivas. Es una de las maneras, quizás la que nos es más familiar, para obtener un mayor nivel de detalle en la representación de los modelos 3D.

Esta información se transmite a la siguiente etapa en unidades discretas conocidas como *batches*.

El resultado del ensamblado da lugar a una secuencia de triángulos, líneas o puntos, en la cual no todos los elementos tienen porque ser procesados. A este conjunto, por lo tanto, se le pueden aplicar dos procesos que aligerarán la carga de trabajo del hardware gráfico.

Por un lado las primitivas pueden ser descartadas mediante el proceso de *clipping*, en el cual se selecciona sólo a aquellas que caen dentro del volumen de visualización (la región visible para el usuario de la escena 3D, conocido también como *view frustum* o pirámide de visión). Y por otro lado, el rasterizador puede descartar también aquellas primitivas cuya cara no esté apuntando hacia el observador, mediante el proceso conocido como *culling*.

Las primitivas que sobreviven a estos procesos son rasterizadas. La rasterización es el proceso mediante el que se determina el conjunto de pixeles que cubren una determinada primitiva. Polígonos, líneas y puntos son rasterizados de acuerdo a unas reglas especificadas para cada tipo de primitiva. El resultado de la rasterización son un conjunto de localizaciones de pixeles, al igual que un conjunto de fragmentos. Es importante recalcar que no hay ninguna relación entre el conjunto de fragmentos generados en la rasterización y el número de vértices que hay en una primitiva. Por ejemplo, un triángulo que ocupe toda la pantalla provocará la creación de miles de fragmentos.

Framebuffer

Los *framebuffer* son dispositivos gráficos que ofrecen una zona de memoria de acceso aleatorio, que representa cada uno de los pixeles de la pantalla.

¿Pixel o fragmento? Habitualmente se usan indistintamente los términos pixel y fragmento para referirse al resultado de la fase de rasterización. Sin embargo existen diferencias importantes entre estos dos términos y considero apropiado aclarar qué es cada uno. Pixel proviene de la abreviatura de "picture element" y representa el contenido del framebuffer en una localización específica, al igual que el color, profundidad y algunos otros valores asociados con esa localización. Un fragmento sin embargo representa el estado potencialmente requerido para actualizar un pixel en particular. En la etapa en la que nos encontramos del pipeline es mucho más apropiado el término fragmento ya que estamos ante pixeles potenciales, contienen toda la información de un pixel pero son candidatos a pixel, no pixeles.

El término *"fragmento"* es usado porque la rasterización descompone cada primitiva geométrica, como puede ser un triángulo, en fragmentos del tamaño de un pixel por cada pixel que la primitiva cubre y al estar discretizándolo, lo fragmenta. Un fragmento tiene asociada una localización para el pixel, un valor de profundidad, y un conjunto de parámetros interpolados como son: el color primario, el color especular, y uno o varios conjuntos de coordenadas de textura. Estos parámetros interpolados son derivados de los parámetros incluidos en los vértices transformados de la primitiva que generó los fragmentos.

Como la rasterización ocurre con todas las primitivas y puede ser que detrás del triángulo de ejemplo hubiera otro triángulo y se generarán fragmentos en la misma posición de pantalla que los que se han generado para el triángulo delantero. Pero tranquilos, como buenos pixeles potenciales que son, guardan información de su profundidad y transparencia, con lo que en las dos etapas siguientes se puede controlar que un fragmento sólo actualice su pixel correspondiente en el *framebuffer* si supera los distintos tests de rasterización que tenemos disponibles.

Scissor's Test

Este test permite restringir el área de dibujado de la pantalla, descartando así, todos aquellos fragmentos que no entren dentro.

Alpha Test

Permite descartar fragmentos comparando el valor de *alpha* de cada fragmento, con un valor constante especificado.

Stencil Test

A partir del uso del *stencil buffer*, hace una comparación con el *framebuffer*, y descarta aquellos fragmentos que no superen la condición especificada. Como si usara una máscara, o una plantilla, para especificar qué se dibuja y qué no.

Interpolación, texturizado y coloreado

Una vez las primitivas han sido rasterizadas en una colección de cero o más fragmentos, la fase de interpolación, texturizado y coloreado se dedica precisamente a eso, a interpolar los parámetros de los fragmentos como sea necesario, realizando una secuencia de operaciones matemáticas y de texturizado que determinan el color final de cada fragmento.

Como complemento a la determinación del color de los fragmentos, esta etapa puede encargarse también de calcular la profundidad de cada fragmento pudiendo descartarlos y así evitar la actualización del correspondiente pixel en pantalla. Debido a que los fragmentos pueden ser descartados, esta etapa devuelve entre uno y cero fragmentos coloreados por cada uno recibido.

Operaciones de Rasterización

La fase en que se ejecutan las operaciones de *rasterización* pone en marcha una secuencia de tests para cada cada fragmento, inmediatamente antes de actualizar el *framebuffer*. Estas operaciones son una parte estándar de *OpenGL* y *Direct3D*, e incluyen: el *scissor test*, *alpha test*, *stencil test* y el *depth test*. En ellos están involucrados el color final del fragmento, su profundidad, su localización, así como su valor de *stencil*.

Si cualquier test falla, esta etapa descarta el fragmento sin actualizar el valor de color del pixel (sin embargo, podría ocurrir una operación de escritura para el valor *stencil*). Pasar el *depth test* puede reemplazar el valor de profundidad del pixel, por el valor de profundidad del fragmento. Despues de los tests, la operación de *blending* combina el color final del fragmento con el valor de color del pixel correspondiente. Finálmente, con una operación de escritura sobre el *framebuffer* se reemplaza el color del pixel, por el color mezclado.

Conclusión

Depth Test

A partir del valor de profundidad del fragmento establece qué fragmento está más cerca de la cámara (podría haber fragmentos por delante) y, en función de la condición especificada lo descarta o no.

Tras concluir esta serie de pasos obtenemos en el *framebuffer*, al fín, la imagen generada a partir de nuestra escena 3D. La cual podrá ser volcada a la pantalla, o usada para algún otro propósito.

Como se puede apreciar, la libertad para influir en el proceso de renderizado en este tipo de *pipeline* está muy limitada. Cualquier transformación sobre los vértices debe hacerse mediante código de la aplicación (siendo la CPU la encargada de ejecutarlo) y se limita a ofrecer los diferentes tests comentados, junto con algun parámetro configurable más, que son los que proveen algo de control sobre la forma en que se renderizan las imagenes. Todo esto es más que suficiente para obtener resultados decentes en muchos tipos de videojuegos. ¿Pero acaso no nos gustan también los gráficos 3D alucinantes?

26.4.5. Programmable-Function Pipeline

Gracias al *programmable-function pipeline* muchas de las operaciones que antes asumía la CPU, definitivamente pasan a manos de la GPU.

La figura 26.13 muestra las etapas de procesamiento de vértices y fragmentos en una GPU con un pipeline programable simplificado. En el, se puede apreciar que se mantienen las mismas etapas que en el *fixed-function pipeline*, pero se añaden tres etapas nuevas en las que los *shader* se encargan de establecer cómo se procesan los vértices, primitivas y fragmentos.

A continuación, en la siguiente sección, se explica someramente cómo funcionan estas nuevas etapas, que se corresponden con cada uno de los tipos de *shader* que existen a día de hoy.

Tipos de Shader

Originalmente los *shaders* sólo realizaban operaciones a nivel de pixel (o fragmento). Los que hoy se conocen como *fragment/pixel shaders*. A lo largo del tiempo se han introducido más tipos de *shader*], por lo que ahora el término *shader* se ha vuelto mucho más genérico, abarcando los tres tipos que se usan hoy en día en la generación de gráficos en tiempo real.

Vertex Shader

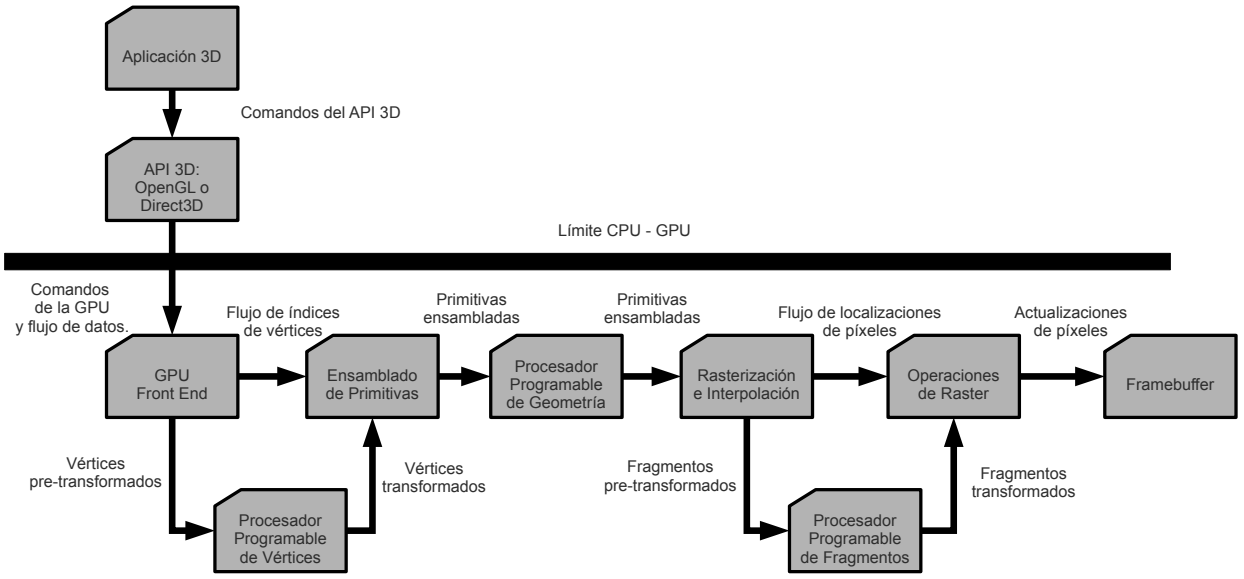


Figura 26.13: Pipeline del hardware gráfico programable simplificado.

El flujo de datos del procesamiento de vértices comienza cargando los atributos de cada vértice (posición, color, coordenadas de textura, etc...) en el procesador de vértices. Este va ejecutando las distintas operaciones secuencialmente para cada vértice hasta que termina. El resultado de esta etapa es un vértice transformado en función de las instrucciónes del *shader*. Después del ensamblaje de la primitiva geométrica y de la rasterización, los valores interpolados son pasados al procesador de fragmentos.

Como ya se ha comentado, los *vertex shader* tienen acceso y pueden modificar los atributos de los vértices. A su vez se permite a cada vértice acceder a lo que se conoce como variables *uniformes*. Estas son variables globales de sólo lectura que permiten acceder a información que no cambia a lo largo de una pasada del material, como pueden ser por ejemplo, la matriz mundo/vista o la delta de tiempo (tiempo pasado entre una vuelta del bucle principal y otra).

Un *vertex shader* recibe como entrada un vértice y devuelve siempre un sólo vértice, es decir, no puede crear nueva geometría.

Cada *vertex shader* afecta a un sólo vértice, es decir, se opera sobre vértices individuales no sobre colecciones de ellos. A su vez, tampoco se tiene acceso a la información sobre otros vértices, ni siquiera a los que forman parte de su propia primitiva.

En el listado de código 26.5 se muestra un ejemplo de este tipo de *shaders*. Cómo se puede ver, la sintaxis es muy parecida a la de un programa escrito en C.

Listado 26.5: Ejemplo de Vertex Shader

```
// Estructura de salida con la información del vertice procesado
struct tVOutput {
    float4  position:   POSITION;
    float4  color   :   COLOR;
};

// Usamos la posición y color del vértice
tVOutput v_color_passthrough(
    float4          position    :   POSITION,
    float4          color       :   COLOR
    uniform float4x4    worldViewMatrix) {
    tVOutput    OUT;
    // Transformación del vértice
    OUT.position = mul(worldViewMatrix, position);
    OUT.color = color;
    return OUT;
}
```

El fragmento de código es bastante explicativo por si mismo porque, básicamente, no hace nada con el vértice. Recibe algunos parámetros (color y posición) y los devuelve. La única operación realizada tiene que ver con transformar la posición del vértice a coordenadas de cámara.

Fragment Shader

Los procesadores de fragmentos requieren del mismo tipo de operaciones que los procesadors de vértices con la salvedad de que tenemos acceso a las operaciones de texturizado. Estas operaciones permiten al procesador acceder a la imagen, o imagenes, usadas de textura y permiten manipular sus valores.

Los *fragment shader* tienen como propósito modificar cada fragmento individual que les es suministrado desde la etapa de *rasterizacion*.

Estos tienen acceso a información como es: la posición del fragmento, los datos interpolados en la *rasterizacion* (color, profundidad, coordenadas de textura), así como a la textura que use la primitiva a la cual pertenece (en forma de variable uniforme, como los *vertex shader*), pudiendo realizar operaciones sobre todos estos atributos.

Al igual que los *vertex shader*, sólo puede procesar un fragmento cada vez y no puede influir sobre, ni tener acceso a, ningún otro fragmento.

Listado 26.6: Ejemplo de Fragment Shader

```
// Estructura de salida con la información del fragmento procesado
struct tFOutput {
    float4 color : COLOR;
};

// Devuelve el color interpolado de cada fragmento
tFOutput f_color_passthrough(
    float4 color : COLOR)
{
```

```
10      tFOutput OUT;
11      // Se aplica el color al fragmento
12      OUT.color = color;
13
14      return OUT;
15  }
```

En el listado de código 26.6 se muestra un ejemplo de este tipo de *shaders*.

Geometry Shader

Este es el más nuevo de los tres tipos de shader. Puede modificar la geometría e incluso generar nueva de forma procedural. Al ser este un tipo de *shader* muy reciente todavía no está completamente soportado por las tarjetas gráficas y aun no se ha extendido lo suficiente.

La etapa encargada de procesar la geometría estaría enclavada entre las etapas de ensamblado de primitivas y la de rasterización e interpolación (ver Figura 26.13).

Este tipo de *shader* recibe la primitiva ensamblada y, al contrario que los *vertex shader*, si tiene conocimiento completo de la misma. Para cada primitiva de entrada tiene acceso a todos los vértices, así como a la información sobre cómo se conectan.

En esta sección no se tratarán, pero cualquiera que lo desee puede encontrar más información al respecto en el capítulo 3 del libro *"Real-Time Rendering"* [8].

26.4.6. Aplicaciones de los Shader

Existen múltiples y diferentes aplicaciones para los shaders. En esta sección se enumeran algunas de las funciones que pueden ayudar a cumplir.

Vertex Skinning

Los vértices de una superficie, al igual que el cuerpo humano, son movidos a causa de la influencia de una estructura esquelética. Como complemento, se puede aplicar una deformación extra para simular la dinámica de la forma de un músculo.

En este caso el *shader* ayuda a establecer cómo los vértices se ven afectados por el esqueleto y aplica las transformaciones en consecuencia.

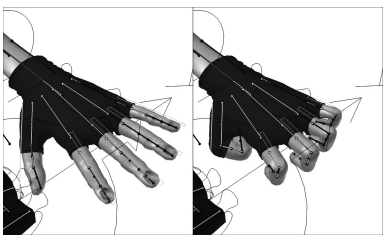


Figura 26.14: Ejemplos de *skinning* (Fuente: Wikipedia).

Vertex Displacement Mapping

Los vértices pueden ser desplazados, por ejemplo verticalmente, en función de los resultados ofrecidos por un algoritmo o mediante la aplicación de un mapa de alturas (una textura en escala de grises), consiguiendo con ello, por ejemplo, la generación de un terreno irregular.

Figura 26.15: Ejemplos de *vertex displacement mapping* (Fuente: Wikipedia).

Screen Effects

Los *shader* pueden ser muy útiles para lograr todo tipo de efectos sobre la imagen ya generada tras el renderizado. Gracias a las múltiples pasadas que se pueden aplicar, es posible generar todo tipo de efectos de post-procesado, del estilo de los que se usan en las películas actuales.

Los *fragment shader* realizan el renderizado en una textura temporal (un *frame-buffer* alternativo) que luego es procesada con filtros antes de devolver los valores de color.

Figura 26.16: Ejemplo de *glow* o *bloom* (Fuente: Wikipedia).

Light and Surface Models

Mediante *shaders* es posible calcular los nuevos valores de color aplicando distintos modelos de iluminación, lo cual involucra parametros como son las normales de las superficies (N), angulo en el que incide la luz (L), angulo de la luz reflejada (R) y el ángulo de visión. De esta forma se pueden conseguir unos resultados finales con

una altísima calidad, aunque habría que tener cuidad con su uso. El conseguir, por ejemplo, luz por pixel implicaría unas operaciones bastante complejas por cada pixel que se vaya a dibujar en pantalla, lo cual, contando los pixeles de una pantalla, pueden ser muchas operaciones.

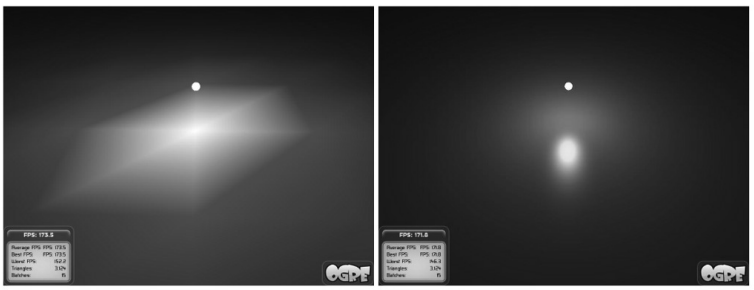


Figura 26.17: Diferencia entre *Per-Vertex Lighting* (izquierda) y *Per-Pixel Lighting* (derecha).

Visual representation improvement

Gracias a estas técnicas es posible lograr que la visualización de los modelos sea mucho mejor, sin incrementar la calidad del modelo. Ejemplo de esta técnica sería el *Normal mapping*, donde a partir de una textura con información sobre el valor de las normales (codificada cada normal como un valor RGB, correspondiente al XYZ de la misma) a lo largo de toda su superficie, permite crear la ilusión de que el modelo cuenta con más detalle del que realmente tiene.

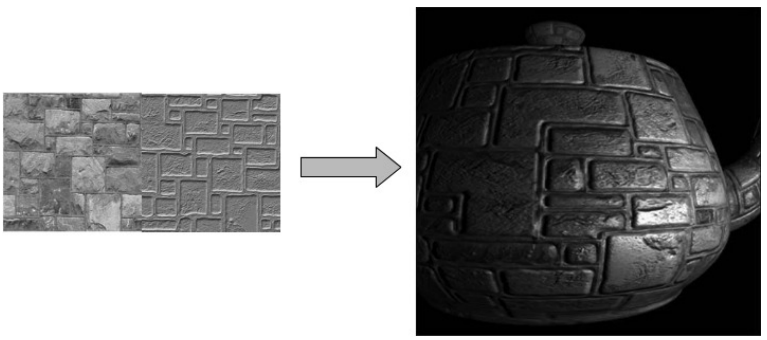


Figura 26.18: Ejemplo de *normal mapping*.

Non-Photorealistic Rendering

Los modelos de iluminación no tienen porque limitarse a imitar el “mundo real”, pueden asignar valores de color correspondientes a mundos imaginarios, como puede ser el de los dibujos animados o el de las pinturas al oleo.

Figura 26.19: Ejemplo de *Toon Shading* (Fuente: Wikipedia).

26.4.7. Lenguajes de Shader

Existe una gran variedad de lenguajes para escribir *shaders* en tiempo real y no parece que vayan a dejar de aparecer más. Estos pueden agruparse en dos categorías: los basados en programas individuales o los que se basan en ficheros de efectos.

La primera aproximación necesita que se cree una colección de ficheros, cada uno de los cuales implementa un *Vertex Shader* o un *Fragment Shader* en una pasada de renderizado. Ejemplos de estos lenguajes son: *Cg*, *HLSL* o *GLSL*. Lo cual conlleva una cierta complejidad en su gestión pues para cada pasada de renderizado, en los efectos complejos, necesita de dos ficheros diferentes.

Para solucionar este problema surgen el siguiente tipo de lenguajes que son aquellos basados ficheros de efectos, de los cuales, el más conocido es el *CgFX* de *NVidia*, y que a su vez es un super conjunto del *Microsoft effect framework*.

Los lenguajes basados en ficheros de efectos permiten que los diferentes *shader* se incluyan en un mismo fichero y, además, introducen dos nuevos conceptos: técnica y pasada. En los cuales pueden ser agrupados los diferentes *shaders* y permiten que el estado del dispositivo gráfico pueda ser definido en cada pasada de renderizado.

Con esta aproximación se consigue que la gestión sea más manejable, mejor dirigida por recursos y mucho más poderosa.

26.5. Desarrollo de shaders en Ogre

Para crear y gestionar los diferentes *shaders* en Ogre se usa la aproximación basada en ficheros de efectos. Los cuales ya se han comentado previamente.

Con el objetivo de hacer lo más sencillo el aprendizaje, primero se explicará cómo montar la escena a mostrar y más tarde se explicará qué es lo que ha pasado.

En estos ejemplos, se usará el lenguaje *Cg*, del que se puede encontrar una documentación muy completa en el libro *"The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics"* [34], que, afortunadamente, puede encontrarse online y de forma gratuita en la página de desarrolladores de NVIDIA:

`http://developer.nvidia.com/object/cg_tutorial_home.html`

También podemos recurrir al capítulo correspondiente a los materiales y *shaders* de *"Ogre 3D 1.7 Beginner's Guide"* [56] que puede ser un buen punto de partida para empezar con este tema.

Poniendo a punto el entorno

Antes de empezar debemos tener claro que para usar los *shader* en Ogre debemos contar con dos tipos de ficheros. Por un lado tendremos los *programas*, o archivos *.cg*, en los que se encontrarán los *shaders* escritos en *Cg*, y por otro lado tendremos los archivos de efectos, o materiales, que definirán cómo se usan nuestros *programas*

Una vez sabido esto, el siguiente paso debería ser dejar claro donde se han colocar los ficheros correspondientes a los shader, para que así puedan ser usados por Ogre. Estos se deben colocar en la ruta:

Log de Ogre

En el fichero de log podremos encontrar información sobre los perfiles y funcionalidad soportada por nuestra tarjeta gráfica, así cómo información sobre posibles errores en la compilación de nuestros materiales o shaders.

```
/[directorio_proyecto_ogre]/media/materials/programs
```

Si queremos usar otro directorio deberemos indicarlo en el archivo *resources.cfg*, bajo la etiqueta *[popular]* por ejemplo (hay que tener cuidado sin embargo con lo que se coloca bajo esa etiqueta en un desarrollo serio).

```
[Popular]
...
FileSystem=../../media/materials/programs/mis_shaders
...
```

Por otro lado, es conveniente tener a mano el fichero de *log* que genera Ogre en cada ejecución, para saber por qué nuestro *shader* hace que todo aparezca blanco (o lo que es lo mismo ¿por qué ha fallado su compilación?).

26.5.1. Primer Shader

Este primer *shader* servirá de toma de contacto para familiarizarnos con algunos de los conceptos que introduce. Es por eso que no hace prácticamente nada, sólo deja que el vértice pase a través de el, modificando únicamente su color y enviándolo a la siguiente etapa.

La escena 3D puede ser cualquiera, siempre y cuando alguna de las entidades tenga asociado el material que se definirá ahora.

Lo primero es indicarle a Ogre alguna información sobre los *shader* que queremos que use, la cual debe estar incluida en el propio archivo de material.

Al definir el *shader* que usará nuestro material hay que indicar al menos:

- El nombre del *shader*
- En qué lenguaje está escrito
- En qué fichero de código está almacenado
- Cómo se llama el punto de entrada al shader
- En qué perfil queremos que se compile

Por último, antes de definir el material en si, hay que indicar también a los shader cuales son aquellos parámetros que Ogre les pasará. En este caso sólo pasamos la matriz que usaremos para transformar las coordenadas de cada vértice a coordenadas de cámara. Es importante definir esta información al principio del fichero de material.

Listado 26.7: Declaración del *vertex shader* en el material

```
// Declaracion vertex shader y lenguaje usado
vertex_program VertexGreenColor cg
{
    // Archivo con los programas
    source firstshaders.cg

    // Punto de entrada al shader
    entry_point v_green_color

    // Perfiles validos
    profiles vs_1_1 arbvp1

    // Parámetros usados
    default_params
    {
        param_named_auto worldViewMatrix worldviewproj_matrix
    }
}
```

Para este caso, el material quedaría tal y como se ve en el listado 26.8. Y queda claro cómo en la pasada se aplican los dos *shader*.

Listado 26.8: Primer Material con shaders

```
vertex_program VertexGreenColor cg
{
    source firstshaders.cg
    entry_point v_green_color
    profiles vs_1_1 arbvp1

    default_params
    {
        param_named_auto worldViewMatrix worldviewproj_matrix
    }
}

fragment_program FragmentColorPassthrough cg
{
    source firstshaders.cg
    entry_point f_color_passthrough
    profiles ps_1_1 arbfp1
}

material VertexColorMaterial
{
    technique
    {
        pass
        {
            vertex_program_ref VertexGreenColor
            {
            }

            fragment_program_ref FragmentColorPassthrough
            {
            }
        }
    }
}
```

Declaración shaders

Para más información sobre la declaración de los *shader*, sería buena idea dirigirse al manual en: http://www.ogre3d.org/-docs/manual/manual_18.html

Los dos programas que definen nuestros primeros *fragment* y *vertex shader* aparecen en los listados 26.9 y 26.10, y los dos deberían incluirse en el fichero *firstshaders.cg* (o en el archivo que queramos), tal y como indicamos en el material.

Listado 26.9: Primer vertex shader

```
// Estructura de salida con la información del vertice procesado
struct tVOutput {
    float4  position:   POSITION;
    float4  color   :   COLOR;
};

// Cambia el color primario de cada vertice a verde
tVOutput v_green(
    float4  position : POSITION,
    uniform float4x4 worldViewMatrix)
{
    tVOutput OUT;
    // Transformamos la posición del vértice
    OUT.position = mul(worldViewMatrix, position);
    // Asignamos el valor RGBA correspondiente al verde
    OUT.color = float4(0, 1, 0, 1);
    return OUT;
}
```

El *vertex shader* recibe cómo único parámetro del vértice su posición, esta es transformada a coordenadas del espacio de cámara gracias a la operación que realizamos con la variable que representa la matriz de transformación.

Por último se asigna el color primario al vértice, que se corresponde con su componente difusa, y que en este caso es verde.

Listado 26.10: Primer fragment shader

```
// Estructura de salida con la información del fragmento procesado
struct tFOutput {
    float4 color : COLOR;
};

// Devuelve el color interpolado de cada fragmento
tFOutput f_color_passthrough(
    float4 color : COLOR)
{
    tFOutput OUT;
    // Se aplica el color al fragmento
    OUT.color = color;

    return OUT;
}
```

El *fragment shader* se limita símplemente a devolver el color interpolado (aunque en este caso no se note) y el resultado debería ser como el mostrado en la Figura 26.20

Cualquiera con conocimientos de C o C++, no habrá tenido mucha dificultad a la hora de entender los dos listado de código anteriores. Sin embargo si habrá notado la presencia de algunos elementos nuevos, propios de Cg. Y es que hay algunas diferencias importantes, al tratarse de un lenguaje tan especializado.

En primer lugar, en Cg no hay necesidad de especificar qué tipos de elementos o librerías queremos usar (como se hace en C o C++ con los `#include`). Automáticamente se incluye todo lo necesario para un programa escrito en este lenguaje.

Por otro lado, tenemos un conjunto completamente nuevo de tipos de datos, que se encargan de representar vectores y matrices.

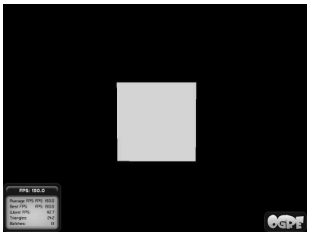


Figura 26.20: Resultado del uso de los primeros *shader*.

En los lenguajes de programación clásicos, los tipos de datos representan magnitudes escalares (`int`, o `float` son ejemplos de ello). Pero, cómo hemos visto en la sección 26.4.5, las GPU están preparadas para tratar con vectores (entiendolos como una colección de magnitudes escalares). Los vectores en C o C++ pueden representarse fácilmente mediante arrays de valores escalares, sin embargo, como su procesamiento es fundamental en el tratamiento de vértices o fragmentos, Cg ofrece tipos predefinidos para estas estructuras.

Para el tratamiento de vectores, podemos usar los siguientes tipos:

```
float2 uv_coord;

float3 position;

float4 rgba_color;
```

O sus equivalentes con la mitad de precisión:

```
half2 uv_coord;

half3 position;

half4 rgba_color;
```

Estos tipos de datos son mucho más eficientes que el uso de un array, por lo que no es recomendable sustituir un `float4 X`, por un `float X[4]`, ya que no son exactamente lo mismo. Cg se encarga de almacenar esta información en una forma, mediante la cual, es posible sacar mucho mejor partido de las características de una GPU a la hora de operar con estos elementos.

Por otro lado, Cg soporta nativamente tipos para representar matrices. Ejemplos de declaraciones serían:

```
float4x4 matrix1;

float2x4 matrix2;
```

Al igual que los vectores, el uso de estos tipos garantiza una forma muy eficiente de operar con ellos.

Otra de las cosas que seguramente pueden llamar la atención, es la peculiar forma en que se declaran los tipos de los atributos pertenecientes a la estructura usada *tVOutput*, o *tFOutput*.

```
float4 position : POSITION;
```

A los dos puntos, y una palabra reservada tras la declaración de una variable (como **POSITION** o **COLOR**), es lo que, en Cg, se conoce como *semántica*. Esto sirve para indicar al *pipeline* gráfico, qué tipo de datos deben llenar estas variables.

Atributos especiales Cg

Para más información sobre ellos se recomienda consultar los capítulos 2 y 3 del muy recomendable *"The Cg Tutorial"* [34]

Es decir, como ya sabemos, los shader son programas que se ejecutan en ciertas etapas del *pipeline*. Por lo tanto, estas etapas reciben cierta información que nosotros podemos usar indicando con la *semántica* cuáles son estos datos. Los únicos sitios donde se pueden usar son en estructuras de entrada o salida (como las de los ejemplos) o en la definición de los parámetros que recibe el punto de entrada (método principal) de nuestro *shader*.

Por último, es necesario hablar brevemente sobre la palabra reservada *uniform*, su significado, y su conexión con lo definido en el fichero de material.

El *pipeline* debe proveer de algún mecanismo para comunicar a los *shader* los valores de aquellos elementos necesarios para conocer el estado de la simulación (la posición de las luces, la delta de tiempo, las matrices de transformación, etc...). Esto se consigue mediante el uso de las variable declaradas como *uniform*. Para Cg, estas son variables externas, cuyos valores deben ser especificados por otro elemento.

En la declaración del *vertex shader* en el material (listado 26.8) se puede ver cómo se declara el parámetro por defecto `param_named_auto worldViewMatrix worldviewproj_matrix`, que nos permite acceder a la matriz de transformación correspondiente mediante un parámetro *uniform*.

26.5.2. Comprobando la interpolación del color

Con este segundo *shader*, se persigue el objetivo de comprobar cómo la etapa de rasterización e interpolación hace llegar al *fragment shader* los fragmentos que cubren cada primitiva con su componente de color interpolado a partir de los valores de color de los vértices.

Para ello es necesario que creemos una escena especialmente preparada para conseguir observar el efecto.

Listado 26.11: Escena definida en Ogre

```
1  void CPlaneExample::createScene(void)
2  {
3      // Creamos plano
4      Ogre::ManualObject* manual = createManualPlane();
5      manual->convertToMesh("Quad");
6
7      // Creamos la entidad
8      Ogre::Entity* quadEnt = mSceneMgr->createEntity("Quad");
9
10     // Lo agregamos a la escena
11     Ogre::SceneNode* quadNode = mSceneMgr->getRootSceneNode()->
           createChildSceneNode("QuadNode");
12
13     quadNode->attachObject(ent);
14 }
15
16 Ogre::ManualObject* CPlaneExample::createManualPlane() {
17     // Creamos un cuadrado de forma manual
18     Ogre::ManualObject* manual = mSceneMgr->createManualObject("
           Quad");
19
20     // Iniciamos la creacion con el material correspondiente al
           shader
21     manual->begin("VertexColorMaterial", Ogre::RenderOperation::
           OT_TRIANGLE_LIST);
22
23     // Situamos vértices y sus correspondientes colores
24     manual->position(5.0, 0.0, 0.0);
25     manual->colour(1, 1, 1);
26
27     manual->position(-5.0, 10.0, 0.0);
28     manual->colour(0, 0, 1);
29
30     manual->position(-5.0, 0.0, 0.0);
31     manual->colour(0, 1, 0);
32
33     manual->position(5.0, 10.0, 0.0);
34     manual->colour(1, 0, 0);
35
36     // Establecemos los indices
37     manual->index(0);
38     manual->index(1);
39     manual->index(2);
40
41     manual->index(0);
42     manual->index(3);
43     manual->index(1);
44
45     manual->end();
```

```
46      return manual;
47  }
```

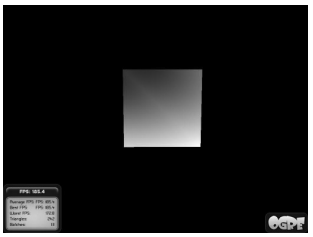


Figura 26.21: El cuadrado aparece con el color de sus cuatro vértices interpolado.

El *vertex shader* lo único que tendrá que hacer es dejar pasar el vértice, pero esta vez, en vez de cambiarle el color dejamos que siga con el que se le ha asignado. Para ello, en el método que define el punto de entrada del *shader* hay que indicarle que recibe como parametro el color del vértice.

Listado 26.12: Segundo vertex shader

```
1  // Estructura de salida con la información del vertice procesado
2  struct tVOutput {
3      float4  position:   POSITION;
4      float4  color   :   COLOR;
5  };
6
7  // Usamos la posición y color del vértice
8  tVOutput v_color_passthrough(
9      float4          position    :   POSITION,
10     float4          color       :   COLOR
11     uniform float4x4    worldViewMatrix) {
12     tVOutput    OUT;
13     // Transformación del vértice
14     OUT.position = mul(worldViewMatrix, position);
15     OUT.color = color;
16     return OUT;
17 }
```

Figura 26.22: Ogro dibujado como si tuviera un mapa de normales encima.

El *fragment shader* podemos dejarlo igual, no es necesario que haga nada. Incluso es posible no incluirlo en el fichero de material (cosa que no se puede hacer con los *vertex shader*).

El resultado obtenido, debería ser el mostrado en la figura 26.21.

Ahora que sabes cómo modificar el color de los vértices. ¿Serías capaz de pintar la entidad típica del Ogro como si de textura tuviera un mapa de normales? Figura 26.22 Tip: Has de usar la normal del vértice para conseguirlo.

26.5.3. Usando una textura

Pintar un modelo con el color de sus vértices puede ser interesante, pero desde luego no es muy impresionante. Por lo tanto con este tercer *shader* usaremos una textura para darle algo de detalle a nuestro plano anterior. Para ello necesitamos acceder a las coordenadas de textura de nuestra entidad cuadrada, por lo que se las añadiremos en nuestro método *createManualPlane*.

Listado 26.13: Creación de un plano al que se le asignan coordenadas de textura

```
1  // Situamos vértices y coordenadas de textura
2  manual->position(5.0, 0.0, 0.0);
3  manual->textureCoord(0, 1);
4  manual->position(-5.0, 10.0, 0.0);
5  manual->textureCoord(1, 0);
6  manual->position(-5.0, 0.0, 0.0);
7  manual->textureCoord(1, 1);
```

```
manual->position(5.0, 10.0, 0.0);
manual->textureCoord(0, 0);
```

Listado 26.14: Declaración del material

```
material TextureMaterial {
    technique
    {
        pass
        {
            vertex_program_ref VertexTexPassthrough
            {
            }

            fragment_program_ref FragmentTexPassthrough
            {
            }

            // Indicamos la textura
            texture_unit
            {
                texture sintel.png
            }
        }
    }
}
```

Para este ejemplo es necesario indicar la textura que se usará y aunque en el capítulo 8 del Módulo 2 ya se habló del tema, será necesario apuntar un par de cosas sobre este nuevo material que aparece en el listado 26.14.

Una vez podemos acceder a las coordenadas de textura del plano, podemos usarlas desde el *vertex shader*. Aunque en este caso nos limitaremos simplemente a pasarla, sin tocarla, al *fragment shader*, tal y como se ve en el listado de código 26.15.

Ahora el *fragment shader* mostrado en el listado 26.16 recibe la coordenada interpolada de textura e ignora el color del vértice. El resultado debería ser el que se ve en la Figura 26.23.

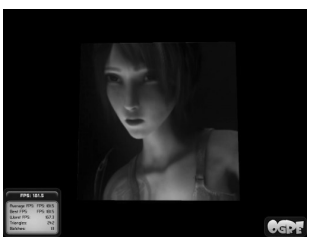


Figura 26.23: El cuadrado aparece con una textura.

Listado 26.15: Tercer vertex shader. Hacemos uso de las coordenadas UV

```
// Vertex shader
struct tVOutput {
    float4  position:   POSITION;
    float2  uv  :   TEXCOORD0;
};

// Usamos la posición y la coordenada de textura
tVOutput v_uv_passthrough(
    float4          position    :   POSITION,
    float2          uv      :   TEXCOORD0,
    uniform float4x4    worldViewMatrix)
{
    tVOutput    OUT;

    // Transformación del vértice
    OUT.position = mul(worldViewMatrix, position);
    OUT.uv = uv;

    return OUT;
}
```

Listado 26.16: Segundo fragment shader. Mapeamos la textura en los fragmentos

```
1  struct tFOutput {
2      float4  color   :   COLOR;
3  };
4
5  // Devuelve el color correspondiente a la textura aplicada
6  tFOutput f_tex_passthrough(
7      float2          uv  : TEXCOORD0,
8      uniform sampler2D   texture)
9  {
10     tFOutput OUT;
11     // Asignamos el color de la textura correspondiente
12     // en función de la coordenada UV interpolada
13     OUT.color = tex2D(texture, uv);
14     return OUT;
15 }
```

En el anterior *shader* no se introdujo ningún concepto nuevo más allá de cómo suministrar el color del vértice al *vertex shader*, sin embargo, ahora nos encontramos con una variable `uniform sampler2D`, un tipo nuevo que no habíamos visto, y el método `tex2D`.

En Cg, un *sampler* se refiere a un objeto externo que se puede muestrear, como es una textura. El sufijo 2D indica que es una textura convencional en 2 dimensiones (existen texturas de 1D y 3D). Para ver los diferentes sampler que soporta Cg puedes ir al capítulo 3 de *"The Cg Tutorial"* [34].

El método `tex2D` se encarga de devolver el color de la textura correspondiente a la coordenada de textura que se le pase como parámetro.

¿Serías capaz de crear un *fragment shader* que mostrara la textura modificada por los valores de color de los vértices como en la Figura 26.24? (Unas pocas líneas más adelante está la respuesta)

Figura 26.24: La textura con una cierta tonalidad rojiza.

Figura 26.25: La textura se muestra como el negativo de una foto.

26.5.4. Jugando con la textura

Esta subsección se limitará a mostrar algunos posibles efectos que se pueden conseguir mediante la modificación del *fragment shader*.

Primero, podríamos intentar devolver el color inverso de la textura que use nuestro modelo, dando un efecto como en el negativo de las fotos.

Esto es muy sencillo de conseguir, sólamente debemos restarle a 1 los valores correspondiente del vector RGBA. Veamos cómo:

Listado 26.17: Efecto negativo fotográfico

```
1  // Estructura de salida con la información del fragmento procesado
2  struct tFOutput {
3      float4 color : COLOR;
4  };
5
6  // Devuelve el color inverso por cada fragmento
7  tFOutput f_tex_inverse(
8      float2          uv  : TEXCOORD0,
9      uniform sampler2D   texture)
10 {
```

```
11      tFOutput OUT;
12      OUT.color = 1 - tex2D(texture, uv);
13      return OUT;
14  }
```

Y ya que estamos modificando el color, quizás en lugar de usar el efecto negativo, tenemos intención de potenciar algún color. En el siguiente ejemplo, daremos un tono rojizo a la textura usada.

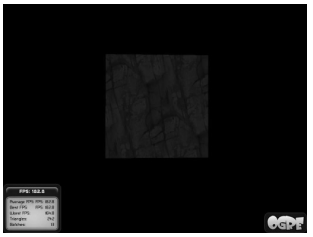


Figura 26.26: Los colores de la textura con su componente roja potenciada.

Listado 26.18: Modificando los colores de la textura con el color rojo

```
1  // Devuelve el color correspondiente a la textura aplicada,
       modificandola para que predomine el rojo
2  tFOutput f_tex_red(
3       float2              uv  : TEXCOORD0,
4       uniform sampler2D   texture)
5  {
6       tFOutput OUT;
7       OUT.color = tex2D(texture, uv);
8       OUT.color.r *= 0.5f;
9       OUT.color.bg *= 0.15f;
10      return OUT;
11 }
```

Como se puede ver, lo único que se ha hecho es multiplicar el componente de color por una cantidad (mayor en el caso del rojo, y menor en el caso del verde y el azúl). A pesar de la sencillez del ejemplo, si has intentado ejecutar este ejemplo con lo aprendido hasta ahora, es probable que no hayas podido.

El motivo es que se introducen dos conceptos nuevos. El primero de ellos se conoce como *swizzling* y consiste en una forma de reordenar los elementos de los vectores, o incluso de acortarlos. Por ejemplo:

```
float4 vec1 = float4(0, 2, 3, 5);

float2 vec2 = vec1.xz; // vec2 = (0, 3)

float scalar = vec1.w; // scalar = 5

float3 vec3 = scalar.xxx; // vec3 = (5, 5, 5)

float4 color = vec1.rgba; // color = (0, 2, 3, 5)
```

Por otro lado, entramos de lleno en el asunto de los perfiles. Los perfiles son una forma de gestionar la funcionalidad de la que disponen los *shader* que programamos, y de esta manera, saber en qué dispositivos podrán funcionar y qué funcionalidad tenemos disponible.

Todos los ejemplos usados hasta ahora usaban los perfiles *vs_1_1* y *ps_1_1* para DirectX 8, y *arbvp1* y *arbfp1* para OpenGL, que son los perfiles más simples a la vez que los más ampliamente soportados. Sin embargo para disponer del *swizzling* es necesario usar los más avanzados *vs_2_0* y *ps_2_0*, que son compatibles con DirectX y OpenGL.

La declaración del *fragment shader* en el material quedará, entonces, como se muetra en el Listado 26.19.

Listado 26.19: Declaración del *fragment shader*

```
1  fragment_program FragmentRedTex cg
2  {
3       source texshaders.cg
4       entry_point f_tex_red
5       profiles ps_2_0 arbfp1
```

```
6 }
```

En el capítulo 2 de *"The Cg Tutorial"* [34], se puede encontrar más información al respecto.

A continuación se presentan algunos ejemplos más de *shaders* que servirán para familiarizarse con la forma en que se trabaja con ellos. Es muy recomendable intentar ponerlos en acción todos y modificar sus parámetros, jugar con ellos, a ver qué sucede.

Listado 26.20: Las funciones trigonómetricas ayudan a crear un patrón de interferencia

```
1 // Devuelve el color de la textura modulado por una funciones
      trigonometricas
2 tFOutput f_tex_interference(
3     float2              uv  : TEXCOORD0,
4     uniform sampler2D   texture)
5 {
6     tFOutput OUT;
7     OUT.color = tex2D(texture, uv);
8     OUT.color.r *= sin(uv.y*100);
9     OUT.color.g *= cos(uv.y*200);
10    OUT.color.b *= sin(uv.y*300);
11    return OUT;
12 }
```

Figura 26.27: Efecto interferencia.

Listado 26.21: Las funciones trigonómetricas ayudan a crear la ondulación en la textura

```
1 // Muestra la imagen modulada por una funcion trigonometrica
2 tFOutput f_tex_wavy(
3     float2              uv  : TEXCOORD0,
4     uniform sampler2D   texture)
5 {
6     tFOutput OUT;
7     uv.y = uv.y + (sin(uv.x*200)*0.01);
8     OUT.color = tex2D(texture, uv);
9     return OUT;
10 }
```

Figura 26.28: Efecto ondulado.

Listado 26.22: La composición del color de la textura en varias posiciones diferentes da lugar a un efecto borroso

```
1 // Como si dibujaramos tres veces la textura
2 tFOutput f_tex_blurry(
3     float2              uv  : TEXCOORD0,
4     uniform sampler2D   texture)
5 {
6     tFOutput OUT;
7     OUT.color = tex2D(texture, uv);
8     OUT.color.a = 1.0f;
9     OUT.color += tex2D(texture, uv.xy + 0.01f);
10    OUT.color += tex2D(texture, uv.xy - 0.01f);
11    return OUT;
12 }
```

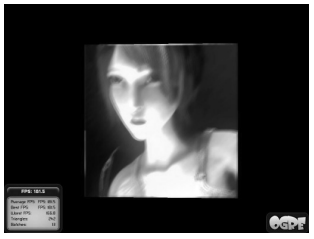


Figura 26.29: Efecto borroso (*blur*).

Listado 26.23: Este efecto se consigue con una combinación del efecto blur con la conversión del color a escala de grises

```
1 // Dibuja la textura como si estuviera grabada en piedra
2 tFOutput f_tex_emboss(
3     float2              uv  : TEXCOORD0,
4     uniform sampler2D   texture)
```

```
5  {
6      float sharpAmount = 30.0f;
7
8      tFOutput OUT;
9      // Color inicial
10     OUT.color.rgb = 0.5f;
11     OUT.color.a = 1.0f;
12
13     // Añadimos el color de la textura
14     OUT.color -= tex2D(texture, uv - 0.0001f) * sharpAmount;
15     OUT.color += tex2D(texture, uv + 0.0001f) * sharpAmount;
16
17     // Para finalizar hacemos la media de la cantidad de color de
           cada componente
18     // para convertir el color a escala de grises
19     OUT.color = (OUT.color.r + OUT.color.g + OUT.color.b) / 3.0f;
20
21     return OUT;
22 }
```

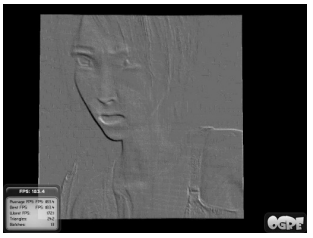


Figura 26.30: Efecto gravado en piedra.

26.5.5. Jugando con los vértices

Antes nos hemos dedicado a jugar un poco con los *fragment shader*, por lo que ahora sería conveniente hacer lo mismo con los *vertex shader*.

Primero usaremos el conocimiento hasta ahora recogido para, a partir de un sólo modelo, dibujarlo dos veces en posiciones distintas. Para conseguirlo se necesitará un material que defina dos pasadas. Una primera en la que el personaje se dibujará en su lugar, y otra en la que será desplazado.

Listado 26.24: Definición de las dos pasadas

```
material CopyObjectMaterial
{
    technique
    {
        pass
        {
            vertex_program_ref VertexColorPassthrough
            {
            }

            fragment_program_ref FragmentTexPassthrough
            {
            }

            texture_unit
            {
                texture terr_rock6.jpg
            }
        }

        pass
        {
            vertex_program_ref VertexDisplacement
            {
            }

            fragment_program_ref FragmentTexPassthrough
            {
            }

            texture_unit
            {
                texture terr_rock6.jpg
```

```
34              }
35          }
36      }
37  }
```

Listado 26.25: Vértice desplazado cantidad constante en eje X

```
1  // Desplazamiento del vértice 10 unidades en el eje X
2  tVOutput v_displacement(
3      float4              position    :   POSITION,
4      uniform float4x4    worldViewMatrix)
5  {
6      tVOutput    OUT;
7      // Modificamos el valor de la posición antes de transformarlo
8      OUT.position = position;
9      OUT.position.x += 10.0f;
10     OUT.position = mul(worldViewMatrix, OUT.position);
11     return OUT;
12 }
```

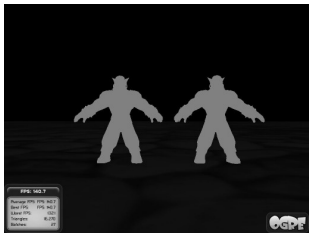


Figura 26.31: Resultado de las dos pasadas.

El anterior era un *shader* muy sencillo, por lo que ahora intentaremos crear nuestras propias animaciones mediante *shaders*.

Para ello necesitamos tener acceso a la delta de tiempo. En la página oficial[1] aparecen listados los diferentes parámetros que Ogre expone para ser accedidos mediante variables `uniform`. En este caso, en la definición del *vertex shader* realizada en el material debemos indicar que queremos tener acceso a la delta de tiempo, como se ve en el Listado 26.26.

Listado 26.26: Acceso a la variable que expone la delta de tiempo

```
1  vertex_program VertexPulse cg
2  {
3      source vertex_modification.cg
4      entry_point v_pulse
5      profiles vs_1_1 arbvp1
6
7      default_params
8      {
9          param_named_auto worldViewMatrix worldviewproj_matrix
10         param_named_auto pulseTime time
11     }
12 }
```

Una vez tenemos accesible el tiempo transcurrido entre dos vueltas del bucle principal, sólo necesitamos pasarlo como parámetro a nuestro *shader* y usarlo para nuestros propósitos. En este caso, modificaremos el valor del eje Y de todos los vértices del modelo, siguiendo una función cosenoidal, como se puede ver en el Listado 26.32.

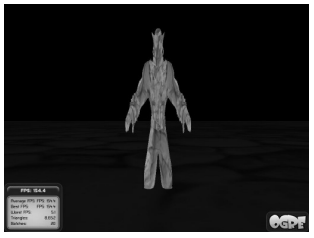


Figura 26.32: Personaje deformado en el eje Y.

Listado 26.27: Combinación de función trigonométrica y delta de tiempo para conseguir simular un movimiento continuo

```
1  // A partir de la delta de tiempo simulamos una señal pulsante para
2  // escalar el modelo en Y
3  tVOutput v_pulse(
4      float4              position    :   POSITION,
5      float2              uv          :   TEXCOORD0,
6      uniform float       pulseTime,
7      uniform float4x4    worldViewMatrix)
```

[1] http://www.ogre3d.org/docs/manual/manual_23.html#SEC128

```
 8 {
 9     tVOutput    OUT;
10
11     OUT.position = mul(worldViewMatrix, position);
12     OUT.position.y *= (2-cos(pulseTime));
13     OUT.uv = uv;
14
15     return OUT;
16 }
```

Sabiendo cómo hacer esto, se tiene la posibilidad de conseguir muchos otros tipos de efectos, por ejemplo en el siguiente se desplaza cada cara del modelo en la dirección de sus normales.

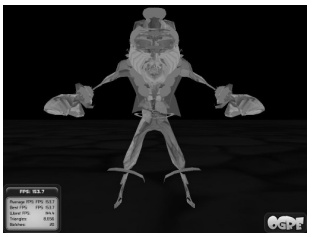


Figura 26.33: Figura con sus vértices desplazados en dirección de sus normales.

Listado 26.28: Uso de la normal y la delta de tiempo para crear un efecto cíclico sobre los vértices

```
1 tVOutput v_extrussion(
2     float4              position    :   POSITION,
3     float4              normal      :   NORMAL,
4     float2              uv          :   TEXCOORD0,
5     uniform float       pulseTime,
6     uniform float4x4    worldViewMatrix)
7 {
8     tVOutput    OUT;
9     OUT.position = position + (normal * (cos(pulseTime)*0.5f));
10    OUT.position = mul(worldViewMatrix, OUT.position);
11    OUT.uv = uv;
12
13    return OUT;
14 }
```

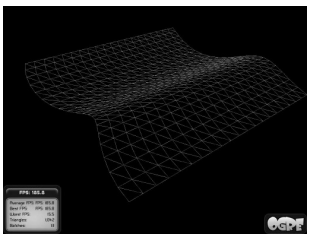


Figura 26.34: Plano con movimiento ondulatorio.

¿Serías capaz de montar una escena con un plano y crear un *shader* que simule un movimiento ondulatorio como el de la Figura 26.34?

26.5.6. Iluminación mediante shaders

Como ejercicio final de esta sección se propone conseguir montar una escena con una luz y un plano y, a partir de ella, conseguir usar un modelo de iluminación por pixel sencillo (como se ve en la Figura 26.17).

En el capítulo 9 del Módulo 2 se habla sobre el tema, a su vez, se puede encontrar una implementación, así como una explicación muy buena sobre iluminación en el capítulo 5 de *"The Cg Tutorial"* [34].

Como pista, o consejo: Se pueden *hardcodear* (poner como constantes) algunos parámetros en el material con el objetivo de simplificar la tarea. En el listado 26.29 se muestran todos los que son necesarios para hacer funcionar el ejercicio de luz por pixel, sólo hay que declararlos usando `param_named`.

Listado 26.29: Ejemplo de declaración de parámetros para ser usados como `uniform` por un *shader*

```
1 default_params
2 {
```

```
3     param_named         globalAmbient float3 0.1 0.1 0.1
4
5     param_named         Ke  float3  0.0 0.0 0.0
6     param_named         Ka  float3  0.0215 0.1745 0.0215
7     param_named         Kd  float3  0.07568 0.61424 0.07568
8     param_named         Ks  float3  0.633 0.727811 0.633
9     param_named         shininess float 76.8
10
11    param_named_auto    eyePosition camera_position_object_space
12
13    param_named_auto    lightPosition
          light_position_object_space 0
14    param_named_auto    lightColorDiffuse   light_diffuse_colour 0
15 }
```

26.6. Optimización de interiores

¿Más velocidad?

Las optimizaciones no sólo aumentan el número de *frames* por segundo, sino que puede hacer interactivo algo que no lo sería sin ellas.

En las dos secciones siguientes se tratarán diferentes maneras de acelerar la representación de escenas. Cabe destacar que estas optimizaciones difieren de las que tienen que ver con el código generado, o con los ciclos de reloj que se puedan usar para realizar una multiplicación de matrices o de vectores. Esta optimizaciones tienen que ver con una máxima: lo que no se ve no debería representarse.

26.6.1. Introducción

Quizá cabe preguntarse si tiene sentido optimizar el dibujado de escenas ya que "todo eso lo hace la GPU". No es una mala pregunta, teniendo en cuenta que muchos monitores tienen una velocidad de refresco de $60Hz$, ¿para qué intentar pasar de una tasa de 70 *frames* por segundo a una de 100?. La respuesta es otra pregunta: ¿por qué no utilizar esos *frames* de más para añadir detalle a la escena? Quizá se podría incrementar el número de triángulos de los modelos, usar algunos *shaders* más complejos o añadir efectos de post-proceso, como *antialiasing* o profundidad de campo.

En este curso las optimizaciones se han dividido en dos tipos: optimización de interiores y optimización de exteriores. En este tema vamos a ver la optimización de interiores, que consiste en una serie de técnicas para discriminar qué partes se pintan de una escena y cuáles no.

Una escena de interior es una escena que se desarrolla dentro de un recinto cerrado, como por ejemplo en un edificio, o incluso entre las calles de edificios si no hay grandes espacios abiertos.

Ejemplos claros de escenas interiores se dan en la saga *Doom* y *Quake*, de *Id Software*, cuyo título *Wolfenstein 3D* fue precursor de este tipo de optimizaciones.

Sin el avance en la representación gráfica que supuso la división del espacio de manera eficiente, estos títulos jamás hubieran sido concebidos. Fue el uso de árboles BSP (2D y 3D, respectivamente) lo que permitió determinar qué parte de los mapas se renderizaba en cada momento.

En general, la optimización de interiores consiste en un algoritmo de renderizado que determina (de forma muy rápida) la oclusiones en el nivel de geometría. Esto diferencia claramente a este tipo de técnicas a las utilizadas para la representación de exteriores, que se basarán principalmente en el procesado del nivel de detalle (LOD (Level-Of-Detail)).

26.6.2. Técnicas y Algoritmos

En su libro, Cormen et all [21] presentan a los algoritmos como una tecnología más a tener en cuenta a la hora de diseñar un sistema. Es decir, no sólo el hardware será determinante, sino que la elección de un algoritmo u otro resultará determinante en el buen rendimiento del mismo. Los videojuegos no son una excepción.

Elección **crucial**

Conocer diferentes algoritmos y saber elegir el más apropiado en cada momento es de suma importancia. Un buen algoritmo puede suponer la diferencia entre poder disfrutar de una aplicación interactiva o no poder hacerlo.

A continuación se presentarán algunas de las técnicas y algoritmos que recopila Dalmau [24] en su libro, que se ha seguido para preparar esta y la siguiente lección, para la representación de escenas de interiores.

Algoritmos basados en Oclusores

Si tuviéramos que determinar qué triángulos dentro de *frustum* ocluyen a otros, la complejidad de llevar a cabo todas las comprobaciones sería de $O(n^2)$, siendo n el número de triángulos.

Listado 26.30: Algoritmo básico basado en oclusores

```
vector<Triangle> occluders = createOccludersSet(sceneTrianles);
vector<Triangle>    others = removeOccluders(sceneTriangles,
                                             occluders);
vector<Triangle>::iterator it;

for (it = others.begin(); it != others.end(); ++it)
{
  if (closerThanFarPlane(*it) &&
      !testOcclusion(*it, occluders))
  {
    (*it)->draw(); // (*it)->addToRendeQueue();
  }
}
```

Dado que esta complejidad es demasiado alta para una aplicación interactiva, se hace necesario idear alguna forma para reducir el número de cálculos. Una forma sencilla es reducir la lista de posibles oclusores. Los triángulos más cercanos a la cámara tienen más posibilidades de ser oclusores que aquellos que están más alejados, ya que estadísticamente estos van a ocupar más área de la pantalla. De este modo, usando estos triángulos como oclusores y los lejanos como ocluidos, se podrá reducir la complejidad hasta casi $O(n)$. Si además se tiene en cuenta que el *frustum* de la cámara tiene un límite (plano *far*), se podrán descartar aun más triángulos, tomando como ocluidos los que estén más allá que este plano.

Este algoritmo podría beneficiarse del *clipping* y del *culling* previo de los triángulos. No es necesario incluir en el algoritmo de oclusores ningún triángulo que quede fuera del *frustum* de la vista actual, tampoco los triángulos que formen parte de las caras traseras se pintarán, ni tendrán que formar parte de los oclusores. La pega es que hay que realizar *clipping* y *culling* por software. Tras esto, se tendrían que ordenar los triángulos en Z y tomar un conjunto de los n primeros, que harán de oclusores en el algoritmo. Computacionalmente, llevar a cabo todas estas operaciones es muy caro. Una solución sería hacerlas sólo cada algunos *frames*, por ejemplo cuando la cámara se moviese lo suficiente. Mientras esto no pasase, el conjunto de oclusores no cambiaría, o lo haría de manera mínima.

Recursividad

Como casi todos los algoritmos de construcción de árboles, el BSP es un algoritmo recursivo. Se dice que una función es recursiva cuando se llama a sí misma hasta que se cumple una condición de parada. Una función que se llama a sí misma dos veces se podrá representar con un árbol binario; una que se llame n-veces, con un árbol n-ario.

Listado 26.31: Actualización de los oclusores

```
const size_t maxOccluders = 300;

newPos = calculateNewCameraPosition();
```

```
4  if (absoluteDifference(oldPos, newPos) > delta)
5  {
6    performClipping(sceneTriangles);
7    performCulling(sceneTriangles);
8    oldPos = newPos;
9    vector<Triangles> occluders =
10     getZOrderedTriangles(sceneTriangles,
11                          maxOccluders);
12 }
```

Algoritmo BSP

Una de las formas para facilitar la representación de escenas interiores en tiempo real es el uso de estructuras de datos BSP (*Binary Space Partition*). Estas estructuras se han utilizado desde juegos como *Doom*, que usaba un árbol BSP de 2 dimensiones, y *Quake*, que fue el primero que usó uno de 3 dimensiones.

Un BSP es un árbol que se utiliza para clasificar datos espaciales, más concretamente triángulos en la mayoría de los casos. La ventaja principal de esta estructura es que se le puede preguntar por una serie de triángulos ordenados por el valor Z, desde cualquier punto de vista de la escena. Estos árboles se usan también para detectar colisiones en tiempo real.

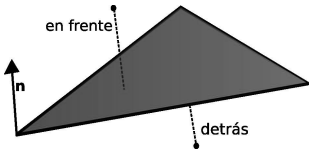


Figura 26.35: Posición con respecto a un plano de normal $\mathbf{n}$.

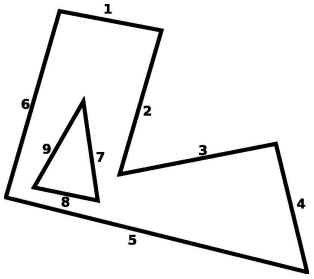


Figura 26.36: Mapa de una escena visto desde arriba.

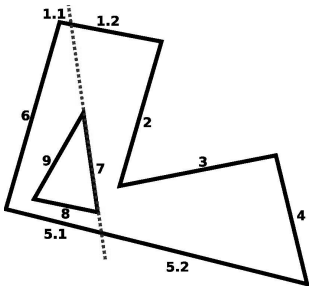


Figura 26.37: Primera división (BSP).

Construcción de estructuras BSP

Un BSP se construye a partir de un conjunto de triángulos, normalmente de la parte estática de una escena, esto es, el mapa del nivel. El algoritmo de construcción es recursivo:

1. Tomar el conjunto completo de triángulos como entrada.
2. Buscar un triángulo que divida a ese conjunto en dos partes más o menos equilibradas.
3. Calcular el plano que corresponde a ese triángulo.
4. Crear un nodo de árbol y almacenar el triángulo y su plano asociado en el mismo.
5. Crear nuevos triángulos a partir de los que queden cortados por este plano.
6. Dividir el conjunto total en dos nuevos conjuntos según queden delante o detrás de la división.
7. Para cada uno de los nuevos conjuntos que aun tengan triángulos (más que un umbral máximo de división dado), volver al paso 2.

Para explicar este algoritmo, se usará una representación de dos dimensiones para simplificar la complejidad. En la figura 26.36 se ve la planta de un nivel. Cada segmento representa a un plano.

Primero se tomará el plano 7 como divisor del nivel, y este dividirá al plano 5 y al 1 en dos (figura 26.37).

El nivel queda dividido en dos nuevos conjuntos, detrás de la división quedan cinco planos (1.1, 6, 5.1, 8 y 9), y delante también (1.2, 2, 3, 4 y 5.2).

En el árbol se crearía un nodo que contendría el plano 7, que es con el que se ha realizado la división. Como los dos subconjuntos nuevos no están vacíos, se crearían de nuevo otros dos nodos (ver figura 26.38).

El algoritmo de división se irá aplicando recursivamente hasta que se hayan realizado todas la divisiones posibles (o se hayan llegado al umbral deseado), tal y como se puede ver en la figura 26.39.

El árbol BSP que se obtiene después de rellenar todos los niveles se muestra en la figura 26.41. Notese cómo el árbol queda bastante equilibrado gracias a una buena elección del plano de división. Un buen equilibrio es fundamental para que pueda desempeñar su labor de manera eficiente, puesto que si el árbol estuviera desequilibrado el efecto sería parecido al no haber realizado una partición espacial.

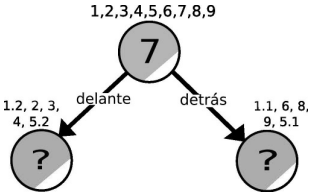


Figura 26.38: Árbol resultante de la primera división (BSP).

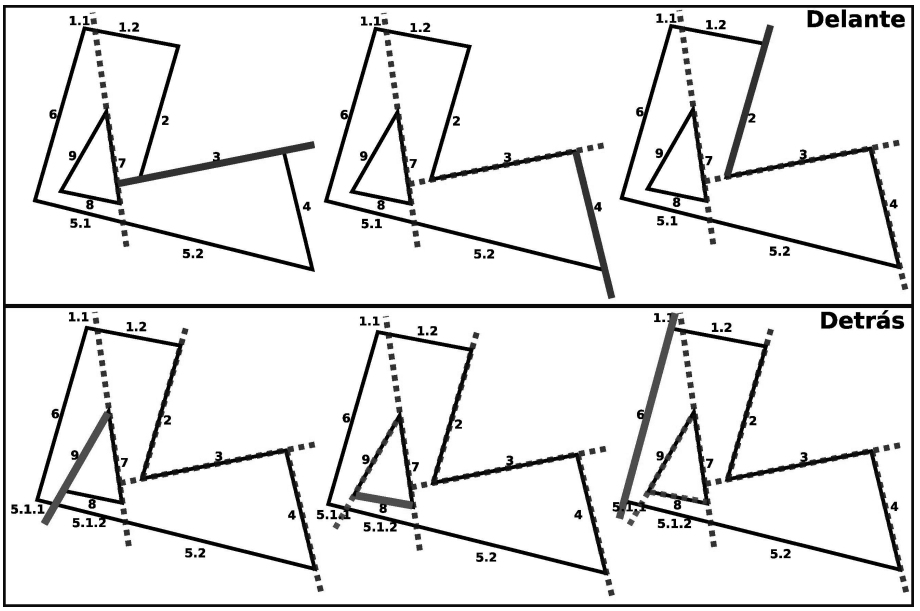


Figura 26.39: Sucesivas divisiones del nivel (BSP)

La elección de un buen triángulo para utilizar como plano de división no es trivial. Es necesario establecer algún criterio para encontrarlo. Un criterio podría ser tomar el triángulo más cercano al centro del conjunto que se tiene que dividir, que podría cortar a muchos otros triángulos, haciendo que creciera rápidamente el número de ellos en los siguientes subconjuntos. Otro criterio podría ser buscar el triángulo que corte a menos triángulos. Uno mejor sería mezclar esta dos ideas.

Ericson [31] analiza algunos problemas relacionados con la elección del plano divisor y propone algunas soluciones simples, parecidas al último criterio propuesto anteriormente.

Otro problema al que hay que enfrentarse es al de la división de los triángulos que resultan cortados por el plano divisor. Cuando se corta un triángulo, este puede dividirse en dos o tres triángulos, siendo mucho más probable que se de la segunda opción. Esto es debido a que la división de un triángulo normalmente genera otro triángulo y un cuadrilátero, que tendrá que ser dividido a su vez en otros dos triángulos (figura 26.40).

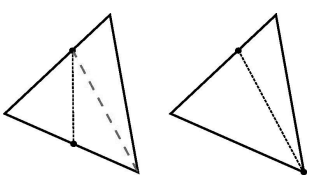


Figura 26.40: División de un triángulo por un plano.

La solución pasa por tomar los vértices del triángulo atravesado como segmentos y hallar el punto de intersección de los mismos con el plano. Con esos puntos será trivial reconstruir los triángulos resultantes.

La estructura de árbol BSP podría estar representada en C++ como en el listado siguiente:

Listado 26.32: Class BSP

```
1 class BSP {
2 public:
3   BSP(vector<Triangle> vIn);
4
5 private:
6   BSP* front;
7   BSP* back;
8   Plane p;
9   Triangle t; // vector<Triangle>vt;
10 };
```

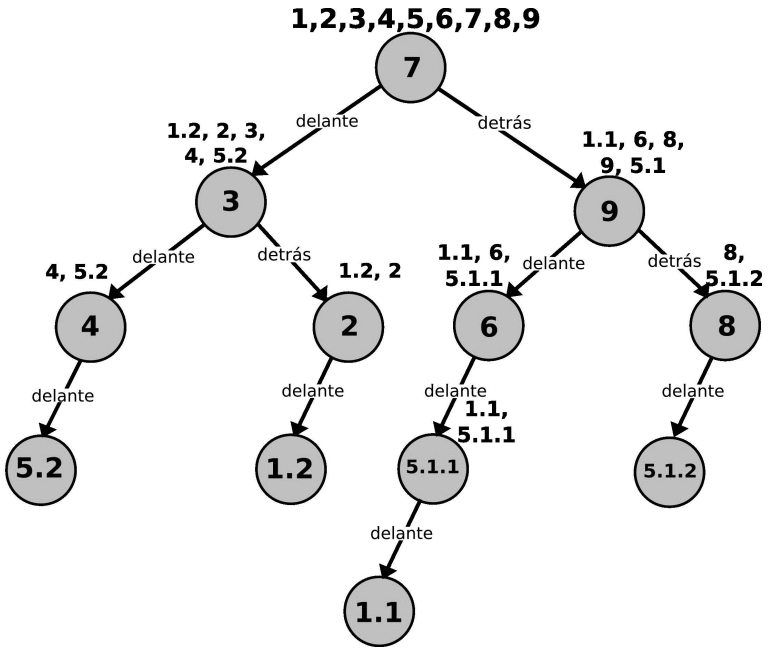


Figura 26.41: Sucesivas divisiones del nivel (BSP)

Su construcción vendría dada por una función como esta:

Listado 26.33: createBSP

```
1 BSP* createBSP(const vector<Triangle>& vIn) {
2
3   BSP* bsp = new BSP;
4
5   bsp->t = getBestTriangle(vt);
6   vtNew = removeTriangleFromVector(vt, t);
7
8   bsp->p = planeFromTriangle(t);
9
```

```
10    vector<Triangle>::iterator it;
11
12    for (it = vt.begin(); vt != vt.end(); ++it) {
13      if (cuts(bsp->p, *it))
14         split(*it, bsp->p);
15    }
16    vector<Triangle> frontSet = getFrontSet(vtNew, t);
17    vector<Triangle> backSet  = getBackSet(vtNew, t);
18
19    bsp->front = createBSP(frontSet);
20    bsp->back = createBSP(backSet);
21
22    return bsp;
23  }
```

Orden dependiente de la vista

La principal ventaja de un BSP es que gracias a él es posible obtener una lista de triángulos ordenados, sea cual sea la vista en la que nos encontremos.

Obsérvese el siguiente listado de código:

Listado 26.34: paintBSP

```
1  void paintBSP(BSP* bsp, const Viewpoint& vp) {
2    currentPos = backOrFront(bsp, vp);
3    if (currentPos == front) {
4      paintBSP(back, vp);
5      bsp->t.addToRenderQueue();
6      paintBSP(front, vp);
7    } else {
8      paintBSP(front, vp);
9      bsp->t.addToRenderQueue();
10     paintBSP(back, vp);
11   }
12 }
```

La función anterior pinta los triángulos (incluidos los que quedan detrás de la vista) en orden, desde el más lejano al más cercano.

Esto era muy útil cuando el *hardware* no implementaba un *Z-buffer*, ya que está función obtenía los triángulos ordenados con un coste linear.

Si cambiamos el algoritmo anterior (le damos la vuelta) recorreremos las caras desde las más cercanas a las más lejanas. Esto sí puede suponer un cambio con el hardware actual, ya que si pintamos el triángulo cuyo valor va a ser mayor en el *Z-buffer*, el resto de los triángulos ya no se tendrán que pintar (serán descartados por el *hardware*).

Clipping Jerárquico

Un BSP se puede extender para usarlo como un sistema de aceleración de *clipping*, quitando los triángulos que queden fuera del *frustum* de la cámara. Lo único que hay que añadir en el árbol durante su construcción en una *bounding box* por cada nodo. Cuanto más se profundice en el árbol, más pequeñas serán, y si el algoritmo de equilibrado de la división es bueno, una *bounding box* contendrá otras dos de un volumen más o menos parecido, equivalente a la mitad de la contenedora.

El algoritmo para recorrer el árbol es muy parecido al anterior, y bastaría con introducir una pequeña modificación.

Listado 26.35: BSPClipping

```
1 void paintBSP(BSP* bsp, const Viewpoint& vp, const Camera& cam) {
2   if ( isNodeInsideFrustum(bsp, cam.getCullingFrustum()) )
3   {
4     // Igual que en el ejemplo anterior
5   }
6 }
```

Detección de la oclusión

También es posible utilizar un árbol BSP para detectar oclusiones. Este uso se popularizó gracias al motor de *Quake*, que utilizaba un nuevo tipo de árbol llamado *leafy*-BSP, donde se utilizaron por primera vez para el desarrollo de un videojuego. Su propiedad principal es la de dividir de manera automática el conjuntos de triángulos entrante en un *array* de celdas convexas.

Este nuevo tipo de árboles son BSPs normales donde toda la geometría se ha propagado a las hojas, en vez de repartirla por todos los nodos a modo de triángulos divisores. De este modo, en un BSP normal, las hojas sólo almacenan el último triángulo divisor.

Para transformar un BSP en un *leafy*-BSP lo que hay que hacer es "agitar" el árbol y dejar caer los triángulos de los nodos intermedios en las hojas (ver figura 26.42)

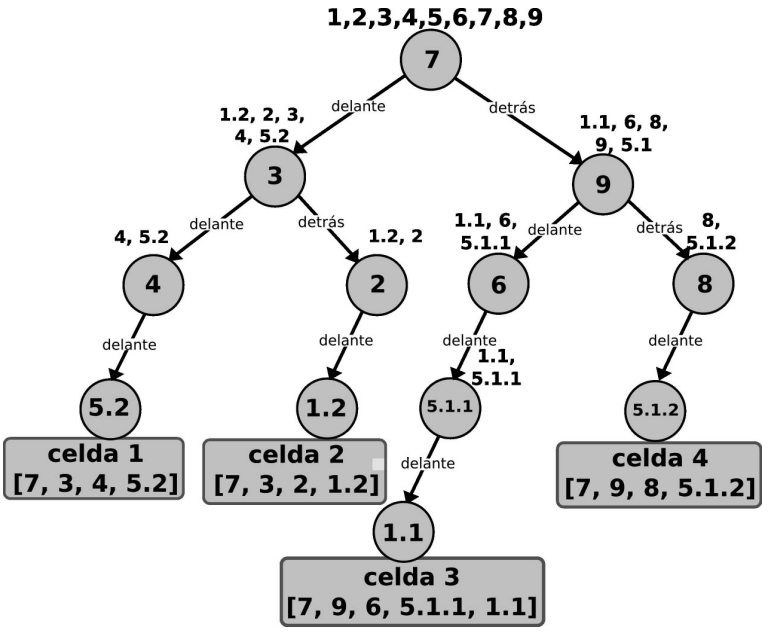


Figura 26.42: Transformación de BSP a *leafy*-BSP.

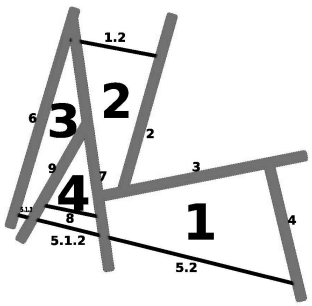


Figura 26.43: Nivel dividido en celdas (l-BSP).

Una vez que el árbol se haya generado, se podrá almacenar la lista de triángulos de cada nodo como una lista de celdas numeradas. Para el ejemplo anterior las celdas se muestran en la figura 26.43.

Cada celda representa una zona contigua y convexa del conjunto de triángulos inicial. Las paredes de las celdas pueden ser o bien áreas ocupadas por geometría del nivel o espacios entre las celdas. Cada espacio abierto debe pertenecer exactamente a dos celdas.

Nótese que el algoritmo anterior convierte cualquier conjunto de triángulos en una lista de celdas y de pasillos que las conectan, que es la parte más complicada.

Es posible precalcular la visibilidad entre las celdas. Para ello se utilizan los pasillos (o portales, aunque diferentes a los que se verán un poco más adelante). Se mandan rayos desde algunos puntos en un portal hacia los demás, comprobándose si llegan a su destino. Si un rayo consigue viajar del portal 1 al portal 2, significa que las habitaciones conectadas a ese portal son visibles mutuamente. Este algoritmo fue presentado por Teller [97].

Esta información sobre la visibilidad se almacenará en una estructura de datos conocida como PVS (Potential Visibility Set), que es sólo una matriz de bits de NxN que relaciona la visibilidad de la fila i (celda i) con la de la columna j (celda j).

Rendering

Para representar los niveles de *Quake III: Arena* se utilizaba un algoritmo más o menos como el que se explica a continuación.

- Se comienza determinando en qué celda se encuentra la cámara (el jugador) utilizando el BSP. Se recorre el árbol desde la raíz, comparando la posición con la del plano divisor para bajar hasta una hoja y elegir una celda determinada.
- Se utiliza el PVS para determinar qué celdas son visibles desde la celda actual, utilizando la matriz de bits (o de booleanos).
- Se *renderizan* las celdas visibles. Se pintan desde el frente hasta el fondo, lo que ayudará al *Z-buffer* a descartar triángulos lo antes posible. Se ordenan las celdas por distancia, se usa su *bounding box* para determinar si quedan dentro del *frustum* para hacer el *clipping* de pantalla y se mandan a *renderizar*.

!Precalcular es la clave!

Los pasos más costosos han sido precalculados, haciendo factible la representación en tiempo real.

Como se ha podido ver, gracias al uso de un árbol *leafy*-BSP se han resuelto casi todos los problemas de determinación de la visibilidad utilizando una estructura precalculada. Esto hace que en el bucle principal del juego no se dedique ningún esfuerzo a computarla. Además, este tipo de estructuras son útiles para determinar colisiones en tiempo real y para ayudar a recorrer los niveles a la IA.

Portal *Rendering*

Otra de las técnicas utilizadas para optimizar la representación de interiores son los portales (Portals). Es un enfoque diferente a los árboles BSP, pero que ofrece una aceleración similar. El motor gráfico *Unreal* demostró su validez utilizando una versión del mismo, y ha ganado adeptos entre los desarrolladores desde entonces. Permite, al igual que los BSPs, representar sólo lo que se ve. En el caso de esta técnica, no se precalcula la visibilidad sino que es computada en tiempo real.

Esta técnica se basa en que los niveles de interiores de un juego están construidos a base de habitaciones interconectadas entres sí por puertas, por ventanas, o en general, por portales. Es de aquí de donde viene su nombre. Para representar estas conexiones entre las diferentes habitaciones será necesario una estructura de datos de grafo no

dirigido. Cada nodo del grafo es una habitación y cada vértice del grafo es un portal. Ya que es posible que en una habitación se encuentren varios portales, es necesario que la estructura de datos permita conectar un nodo con varios otros, o que dos nodos estén conectados por dos vértices.

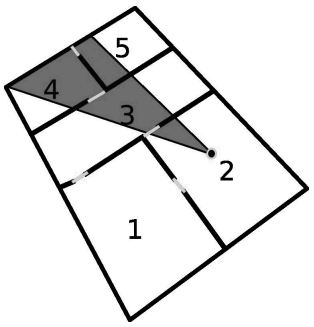


Figura 26.44: Mapa de habitaciones conectadas por portales

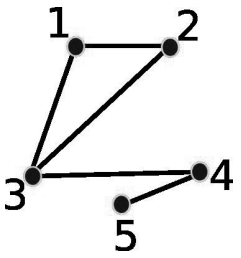


Figura 26.45: Grafo que representa las conexiones del mapa de habitaciones.

Al contrario que los BSP, la geometría de un nivel no determina de manera automática la estructura de portales. Así, será necesario que la herramienta que se use como editor de niveles soporte la división de niveles en habitaciones y la colocación de portales en los mismos. De esto modo, la creación de la estructura de datos es un proceso manual. Estas estructuras no almacenan datos precalculados sobre la visibilidad; esta se determinará en tiempo de ejecución.

El algoritmo de *renderizado* comienza por ver dónde se encuentra la cámara en un momento dado, utilizando los *bounding volumes* de cada habitación para determinar dentro de cuál está posicionada. Primero se pintará esta habitación y luego las que estén conectadas por los portales, de forma recursiva, sin pasar dos veces por un mismo nodo (con una excepción que se verá en el siguiente apartado). Lo complejo del algoritmo es utilizar los portales para hacer *culling* de la geometría.

Es necesario detectar qué triángulos se pueden ver a través de la forma del portal, ya que normalmente habrá un gran porcentaje no visible, tapados por las paredes que rodean al mismo. Desde un portal se puede ver otro, y esto tendrá que tenerse en cuenta al calcular las oclusiones. Se utiliza una variante de la técnica de *view frustum* (que consiste en descartar los triángulos que queden fuera de un frustum, normalmente el de la cámara), que Dalmau llama *portal frustum*. El *frustum* que se utilizará para realizar el *culling* a nivel de portal tendrá un origen similar al de la cámara, y pasará por los vértices del mismo. Para calcular las oclusiones de un segundo nivel en el grafo, se podrá obtener la intersección de dos o más *frustums*.

Portal

¿Tendrá algo que ver esta técnica con algún juego de Valve™? Gracias a ella el concepto de una de sus franquicias ha sido posible.

Un portal puede tener un número de vértices arbitrario, y puede ser cóncavo o convexo. La intersección de dos portales no es más que una intersección 2D, en la que se comprueba vértice a vértice cuáles quedan dentro de la forma de la recursión anterior. Este algoritmo puede ralentizar mucho la representación, puesto que el número de operaciones depende del número de vértices, y la forma arbitraria de los portales no ayuda a aplicar ningún tipo de optimizaciones.

Luebke y Jobes [59] proponen que cada portal tenga asociada un *bounding volume*, que simplificará enormemente los cálculos. Este *bounding volume* rodea a todos los vértices por portal, lo que hará que el algoritmo de como visibles algunos triángulos que no lo son. La pérdida de rendimiento es mínima, y más en el *hardware* actual donde probablemente cada habitación esté representada como un *array* de triángulos.

Efectos ópticos utilizando portales

Una de las ventajas principales de utilizar la técnica de portales en comparación con la de BSP es que se pueden implementar efectos de reflexión y de transparencia, usando el algoritmo central de la misma. Para llevar a cabo este tipo de efectos, lo único que hay que añadir a cada portal es su tipo, por ejemplo los portales podrían ser del tipo normal, espejo, transparencia, o de cualquier otro efecto que se pueda llevar a cabo a través de este tipo de estructuras.

Espejos

Usando portales, poner espejos en la escena tiene un coste gratuito, excepto porque supone representar dos veces la misma habitación.

Listado 26.36: Ejemplo de representación de un portal

```
enum portalType {
  NORMAL,
  MIRROR,
  TRANPARENT,
  INTERDIMENSIONAL,
  BLACK_VOID
};

struct portal {
  vector<Vertex3D*> vertexes_;
  portalType type_;
  Room* room1;
  Room* room2;
};
```

A la hora de representar un portal, podría discriminarse por el tipo, utilizando la técnica adecuada según corresponda.

Listado 26.37: Ejemplo de elección de tipos de portales

```
switch(type_) {
case NORMAL:
  // Algoritmo Normal
  break;

case MIRROR:
  // Calcular la cámara virtual usando el plano de soporte del
     portal
  //
  // Invertir la view-matrix
  //
  // Pintar la habitación destino
  //
  // Pintar la geometría del portal de forma
  // translúcida con algo de opacidad si se desea
  break;

case TRANSPARENT:
  // Pintar de forma normal la habitación que corresponda
  //
  // Pintar la geometría del portal de forma
  // translúcida con algo de opacidad si se desea
  break;

case INTERDIMENSIONAL:
  // Modificar los vértices del array con una función sinuidal
  // Pintarlo
  // Añadir colores chillones a la opacidad del portal.
  break;

case BLACK_VOID:
  // Modificar la geometría para que la habitación
  // destino parezca estirada hacia un agujero negro
  //
```

```
34    // Pintar un borde brillante en los vértices de la forma del
          portal.
35    break;
36  }
```

¿Se le ocurre algún tipo de efecto más o alguna otra forma de aprovechar las características de los portales?

Mapas de Oclusión Jerárquicos (HOM)

Esta técnica, al igual que la de portales, computa la oclusiones en tiempo real durante la ejecución del bucle principal. La ventaja principal de esta técnica es que no es necesario preprocesar la geometría del nivel de ningún modo. Además, otra ventaja de HOM (Hierarchical Occlusion Maps) frente a BSP o portales es que permite utilizar geometría tanto estática como dinámica de manera indiferente.

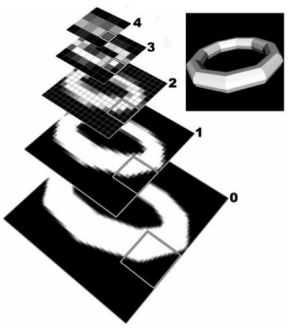


Figura 26.46: HOM. Jerarquía de imágenes (ZHANG).

HOM [106] está basado en una jerarquía de mapas de oclusión. Cada uno de ellos será de la mitad de tamaño que el anterior. Se comienza con una representación a pantalla completa de nuestra escena en blanco y negro. Tras esto se calculan una serie de *mipmaps*, donde cada $2x2$ píxeles se transforman en uno de la nueva imagen. Este algoritmo es muy sencillo si se comienza con una imagen cuadrada potencia de dos. Estas imágenes son las que forman la jerarquía de mapas de oclusión. El mapa de oclusión no contiene la escena completa, sólo un conjunto de posibles oclusores, elegidos con algún método parecido al explicado en el primer algoritmo.

En el bucle principal se pintará objeto por objeto, utilizando algún tipo de estructura no lineal, como un *octree* o un *quadtree*. Para cada objeto se calcula un *bounding rectangle* alineado con la pantalla. Después se toma la imagen (el nivel HOM) con un tamaño de píxel aproximadamente similar al mismo. Este rectángulo puede caer en una zona completamente blanca, y habrá que hacer más comprobaciones (existe un *full-overlap*, el objeto que comprobamos está completamente delante o detrás), puede caer en una zona negra, y se tendrá que pintar, o puede caer en una zona gris, caso en el que habrá que consultar con una imagen de mayor resolución.

Cuando el rectángulo cae en una zona con blanca, es necesario hacer una comprobación sobre los valores Z para comprobar si el objeto está delante o detrás. Esto se consigue con un DEB (Deep Estimation Buffer)[2], que no es más que un *Z-buffer* construido por software, utilizando los posibles oclusores. El DEB almacena la información resultante de crear las *bounding boxes* de los oclusores y almacenar a modo de píxel el valor más lejano (al contrario que un *Z-buffer* normal) para cada posición de la pantalla.

El algoritmo completo podría describirse como sigue:

- Seleccionar un buen conjunto de oclusores. Se descartarán objetos pequeños, o con muchos polígonos, y los objetos redundantes. Es posible colocar objetos falsos que no se pintarán como oclusores a mano, para mejorar el conjunto de manera premeditada.
- En ejecución, se seleccionan los primeros N oclusores más cercanos.

[2] *Buffer* de Estimación de Profundidad

- Calcular el HOM en función de estos objetos. Con la función de *render-to-texture* se crea la primera imagen. Las demás, por software o utilizando otras texturas y alguna función de *mipmapping* de la GPU.
- Mientras se recorre el *scene-graph* se comparan estos objetos con el mapa de oclusión. Si caen en una zona blanca, se comprueban contra el DEB; si caen en una zona negra, se pinta; y si caen en una zona gris, será necesario usar una imagen de mayor resolución.

Dalmau afirma que con esta técnica se evita pintar de media entre un 40 % y un 60 % de toda la geometría entrante.

Enfoques híbridos

En los videjuegos se suele utilizar la combinación de técnicas que más beneficio brinde al tipo de representación en tiempo real a la que se esté haciendo frente. De nuevo, Dalmau propone dos aproximaciones híbridas.

Equilibrio

Utilizar lo mejor de cada una de las técnicas hace que se puedan suplir sus debilidades.

Portal-Octree

En un juego donde el escenario principal está repleto de habitaciones y cada una de ellas está llena de objetos, una aproximación de través de un BSP quizá no sería la mejor idea. No sólo porque este tipo de estructuras está pensado principalmente para objetos estáticos, sino porque un árbol BSP suele extenderse muy rápido al empezar a dividir el espacio.

Si además el juego requiere que se pueda interactuar con los objetos que hay en cada habitación, el BSP queda descartado para almacenar los mismos. Quizá utilizar la técnica de portales pueda usarse para las habitaciones, descartando así algo de geometría del nivel. Aun así la gran cantidad de objetos haría que fueran inmanejables.

Una posible solución: utilizar portales para representar las habitaciones del nivel, y en cada habitación utilizar un *octree*.

Quadtree-BSP

Hay juegos que poseen escenarios gigantestos, con un área de exploración muy grande. Si se enfoca la partición de este tipo de escenas como la de un árbol BSP, el gran número de planos de división hará que crezca la geometría de manera exponencial, debido a los nuevos triángulos generados a partir de la partición.

Una forma de afrontar este problema es utilizar dos estructuras de datos. Una de ellas se usará para realizar una primera división espacial de la superficie (2D, un Quadtree, por ejemplo) y la otra para una división más exhaustiva de cada una de esas particiones. De esto modo, se podrá utilizar un Quadtree donde cada nodo contiene un BSP.

De este modo, se pueden utilizar las características especiales de cada uno de ellos para acelerar la representación. En un primer paso, el Quadtree facilitará la determinación de la posición global de una manera muy rápida. Una vez que se sepa en qué parte del escenario se encuentra la acción, se tomará el BSP asociado a la misma y se procederá a su representación como se mostró en el apartado anterior.

Este tipo de representaciones espaciales más complejas no son triviales, pero a veces son necesarias para llevar a cabo la implementación exitosa de un videojuego.

En el siguiente capítulo ese introducirán los *quadtrees*.

¿Recuerda alguno de estos tipo de escenas en los últimos videojuegos a los que ha jugado?

Tests asistidos por *hardware*

Las tarjetas gráficas actuales proveen de mecanismos para llevar a cabo los cálculos de detección de la oclusión por hardware. Estos mecanismos consisten en llamadas a funciones internas que reducirán la complejidad del código. El uso de estas llamadas no evitará la necesidad de tener que programar pruebas de oclusión, pero puede ser una ayuda bastante importante.

Hardware

El uso del hardware para realizar los tests de oclusión es el futuro, pero eso no quita que se deban conocer las técnicas en las que se basa para poder utilizarlo de manera efectiva.

La utilización del hardware para determinar la visibilidad se apoya en pruebas sobre objetos completos, pudiendo rechazar la inclusión de los triángulos que los forman antes de entrar en una etapa que realice cálculos sobre los mismos. Así, las GPUs actuales proveen al programador de llamadas para comprobar la geometría de objetos completos contra el *Z-buffer*. Nótese como estas llamadas evitarán mandar estos objetos a la GPU para se pintados, ahorrando las transformaciones que se producen antes de ser descartados. Además, como retorno a dichas llamadas se puede obtener el número de píxeles que modificría dicho objeto en el *Z-buffer*, lo que permitiría tomar decisiones basadas en la relevancia del objeto en la escena.

Cabe destacar que si se usa la geometría completa del objeto, mandando todos los triángulos del mismo al test de oclusión de la GPU, el rendimiento global podría incluso empeorar. Es algo normal, puesto que en una escena pueden existir objetos con un gran número de triángulos. Para evitar este deterioro del rendimiento, y utilizar esta capacidad del hardware en beneficio propio, lo más adecuado es utilizar *bounding-boxes* que contengan a los objetos. Una caja tiene tan solo 12 triángulos, permitiendo realizar tests de oclusión rápidos y bastante aproximados. Es fácil imaginarse la diferencia entre mandar 12 triángulos o mandar 20000.

Además, si las pruebas de oclusión para todos los objetos se llevan a cabo de forma ordenada, desde los más cercanos a los más lejanos, las probabilidades de descartar algunos de ellos aumentan.

Como ejemplo, uno tomado de las especificaciones de *occlusion query* de las extensiones de OpenGL [69].

Listado 26.38: Oclusión por hardware

```
GLuint queries[N];
GLuint sampleCount;
GLint  available;
GLuint bitsSupported;

// Comprobar que se soporta la funcionalidad
glGetQueryiv(GL_QUERY_COUNTER_BITS_ARB, &bitsSupported);
if (bitsSupported == 0) {
  // Representar sin test de oclusion...
}

glGenQueriesARB(N, queries);
...

// Antes de este punto, renderizar los oclusores mayores
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glDepthMask(GL_FALSE);

```

```
19 // tambien deshabilitar el texturizado y los shaders inutiles
20 for (i = 0; i < N; i++) {
21   glBeginQueryARB(GL_SAMPLES_PASSED_ARB, queries[i]);
22   // renderizar la bounding box para el objecto i
23   glEndQueryARB(GL_SAMPLES_PASSED_ARB);
24 }
25
26 glFlush();
27
28 // Hacer otro trabajo hasa que la mayoria de las consultas esten
       listas
29 // para evitar malgastar tiempo
30 i = N*3/4; // en vez de N-1, para evitar que la GPU se ponga en
       ïdle"
31 do {
32   DoSomeStuff();
33   glGetQueryObjectivARB(queries[i],
34                         GL_QUERY_RESULT_AVAILABLE_ARB,
35                         &available);
36 } while (!available);
37
38 glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
39 glDepthMask(GL_TRUE);
40
41 // habilitar otro estado, como el de texturizado
42 for (i = 0; i < N; i++) {
43   glGetQueryObjectuivARB(queries[i], GL_QUERY_RESULT_ARB,
44                          &sampleCount);
45   if (sampleCount > 0) {
46     // representar el objecto i
47   }
48 }
```

26.6.3. Manejo de escenas en OGRE

La representación de escenas en OGRE tiene su base en un grafo de escena (*Scene Graph*) un tanto particular. En otros motores gráficos, el grafo de escena está completamente ligado al tipo nodo del mismo. Esto hace que exista un completo acoplamiento entre grafo y nodos, haciendo muy difícil cambiar el algoritmo de ordenación una vez realizada la implementación inicial.

Para solucionar este problema, OGRE interactúa con el grafo de escena sólo mediante su firma (sus métodos públicos), sin importar cómo se comporte internamente. Además, OGRE tan sólo utiliza este grafo como una estructura, ya que los nodos no contienen ni heredan ningún tipo de funcionalidad de control. En vez de eso, OGRE utiliza una clase *renderable* de donde se derivará cualquier tipo de geometría que pueda contener una escena. En el capítulo de fundamentos, se vio la relación entre un tipo *renderable* y un *Movable Object*, que estará ligado a un nodo de escena. Esto quiere decir que un nodo de escena puede existir sin tener ninguna capacidad representativa, puesto que no es obligatorio que tenga ligado ningún `MovableObject`. Con esto se consigue que los cambios en la implementación de los objetos *rendarables* y en la implementación del grafo de escena no tengan efectos entre ellos, desacoplando la implementación de los mismos.

Es posible incluso ligar contenido definido por el programador a un nodo de escena, implementando una interfaz muy sencilla. Gracias a esto, se podrían por ejemplo añadir sonidos ligados a ciertas partes de una escena, que se reproducirían en una parte determinada de la misma.

BSP

El soporte de OGRE para BSP es histórico. Se usa sólo para cargar mapas de Quake3 y este es el único casa para el que se recomienda usarlos.

El futuro de OGRE

Los gestores de escenas de OGRE están evolucionando. Utilizar el gestor por defecto es una buena garantía de que será compatible con las siguientes versiones.

El grafo de escena que se utiliza en OGRE se conoce como *scene manager* (gestor de escena) y está representado por la clase `Scene-Manager`. El interfaz de la misma lo implementan algunos de los gestores de escena que incluye OGRE, y también algunos otros desarrollados por la comunidad o incluso comerciales.

En OGRE es posible utilizar varios gestores de escena a la vez. De esto modo, es posible utilizar uno de interiores y de exteriores a la vez. Esto es útil por ejemplo cuando desde un edificio se mira por una ventana y se ve un terreno.

Un gestor de escena de OGRE es responsable entre otras cosas de descartar los objetos no visibles (*culling*) y de colocar los objetos visibles en la cola de *renderizado*.

Interiores en OGRE

El gestor de escena para interiores de OGRE está basado en BSP. De hecho, este gestor se utiliza con mapas compatibles con *Quake 3*. Hay dos formas de referirse a este gestor, la primera como una constante de la enumeración `Ogre::SceneType` (`ST_INTERIOR`) y otra como una cadena (`"BspSceneManager"`) que se refiere al nombre del *Plugin* que lo implementa. La forma más moderna y la preferida es la segunda.

En la línea [9] se crea el `SceneManager` de BSP, y en la línea [30] se carga el mapa y se usa como geometría estática.

Cabe destacar que para poder navegar por los mapas BSP de manera correcta hay que rotar 90 grados en su vector **pitch** y cambiar el vector **up** de la cámara por el eje Z.

Listado 26.39: Ejemplo de carga de BSP en OGRE

```
1  _root = new Ogre::Root();
2
3  if(!_root->restoreConfig()) {
4    _root->showConfigDialog();
5    _root->saveConfig();
6  }
7
8  Ogre::RenderWindow* window = _root->initialise(true,"BSP");
9  _sceneManager = _root->createSceneManager("BspSceneManager");
10
11 Ogre::Camera* cam = _sceneManager->createCamera("MainCamera");
12 cam->setPosition(Ogre::Vector3(0,0,-20));
13 cam->lookAt(Ogre::Vector3(0,0,0));
14 cam->setNearClipDistance(5);
15 cam->setFarClipDistance(10000);
16 // Cambiar el eje UP de la cam, los mapas de Quake usan la Z
17 // Hacer los ajustes necesarios.
18 cam->pitch(Ogre::Degree(90));
19 cam->setFixedYawAxis(true, Ogre::Vector3::UNIT_Z);
20
21 Ogre::Viewport* viewport = window->addViewport(cam);
22 viewport->setBackgroundColour(Ogre::ColourValue(0.0,0.0,0.0));
23 double width = viewport->getActualWidth();
24 double height = viewport->getActualHeight();
25 cam->setAspectRatio(width / height);
26
27 loadResources();
28
29 [...]
30
31 _sceneManager->setWorldGeometry("maps/chiropteradm.bsp");
```

Se propone crear una escena utilizando un mapa BSP. Se sugiere utilizar el archivo *pk0* que se distribuye con OGRE. Pruebe a navegar por él con y sin el *fix* de la cámara. Añada alguna forma de mostrar los FPS. ¿En qué partes del mapa y en qué condiciones aumenta esta cifra? ¿Por qué?

26.7. Optimización de Exteriores

26.7.1. Introducción

Las diferencias entre una escena de interiores y una de exteriores son evidentes. Mientras una escena de interiores se dará en entornos cerrados, con muchas paredes o pasillos que dividen en espacio en habitaciones, una escena de exteriores normalmente no tiene ningún límite que no esté impuesto por la naturaleza. Si bien es cierto que es una escena de este tipo, por ejemplo, pudiesen existir colinas que se tapasen unas a las otras, si estamos situados frente a algunas de ellas en una posición muy lejana, el número de triángulos que se deberían representar sería tan elevado que quizá ningún hardware podría afrontar su renderizado.

Está claro que hay que afrontar la representación de exteriores desde un enfoque diferente: hacer variable el nivel de detalle LOD). De este modo, los detalles de una montaña que no se verían a cierta distancia no deberían renderizarse. En general, el detalle de los objetos que se muestran grandes en la pantalla (pueden ser pequeños pero cercanos), será mayor que el de los objetos menores.

Si bien el nivel de detalle es importante, tampoco se descarta el uso de oclusiones en algunos de los algoritmos que se presentarán a continuación, siguiendo de nuevo la propuesta de Dalmau. Se comenzará haciendo una pasada rápida por algunas de las estructuras de datos necesarias para la representación de exteriores eficiente.

26.7.2. Estructuras de datos

Uno de los principales problemas que se tienen que resolver para la representación de exteriores es la forma de almacenar escenas compuestas por grandes extensiones de tierra.

Las estructuras de datos utilizadas tendrán que permitir almacenar muchos datos, computar de manera eficiente el nivel de detalle necesario y permitir que la transición entre diferentes dichos niveles sea suave y no perceptible.

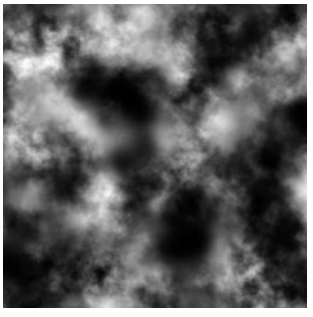


Figura 26.47: Mapa de altura (WIKIPEDIA - PUBLIC DOMAIN)

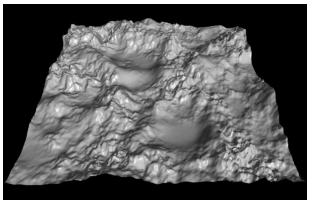


Figura 26.48: Mapa de altura renderizado (WIKIMEDIA COMMONS)

Mapas de altura

Los mapas de altura (*heigtfields* o *heightmaps*) han sido utilizados desde hace mucho tiempo como forma para almacenar grandes superficies. No son más que imágenes en las que cada uno de sus píxeles almacenan una altura.

Cuando se empezó a usar esta técnica, las imágenes utilizaban tan solo la escala de grises de *8-bit*, lo que suponía poder almacenar un total de 256 alturas diferentes.

Los mapas de altura de hoy en día pueden ser imágenes de *32-bit*, lo que permite que se puedan representar un total de 4.294.967.296 alturas diferentes, si se usa el canal *alpha*.

Para transformar uno de estos mapas en una representación 3D es necesario hacer uso de un vector de 3 componentes con la escala correspondiente. Por ejemplo, si se tiene un vector de escala $s = (3, 4, 0{,}1)$ quiere decir que entre cada uno los píxeles del eje X de la imagen habrá 3 unidades de nuestra escena, entre los de Y habrá 4, y que incrementar una unidad el valor del píxel significará subir $0{,}1$ unidades. La posición varía según cómo estén situados los ejes.

Las ventajas de utilizar un mapa de altura es que se pueden crear con cualquier herramienta de manipulación de imágenes y que se pueden almacenar directamente en memoria como *arrays* de alturas que se transformarán en puntos 3D cuando sea necesario, liberando de este modo mucha memoria.

Las principal desventaja viene de que cada píxel representa una sola altura, haciendo imposible la representación de salientes o arcos. Normalmente todo este tipo de detalles tendrán que añadirse en otra capa.

Quadtrees

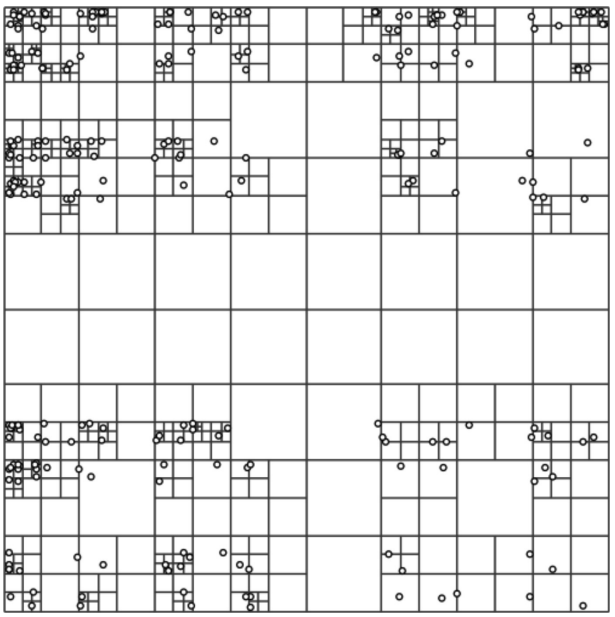


Figura 26.49: Representación de un *Quadtree* (WIKIMEDIA COMMONS - DAVID EPPSTEIN)

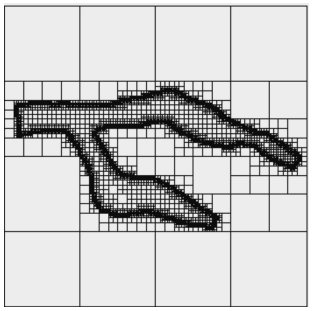


Figura 26.50: División de un terreno con un *quadtree* (QIUHUA LIANG)

Un *quadtree* es un árbol donde cada nodo tendrá exactamente cuatro hijos, así, se dice que es un árbol 4-ario. Un *quadtree* divide un espacio en cuatro partes iguales por cada nivel de profundidad del árbol (figura 26.49).

Un *quadtree* permite que ciertas áreas del terreno se puedan representar con más detalle puesto que es posible crear árboles no equilibrados, utilizando más niveles donde sea necesario. Una aproximación válida es comenzar con un mapa de altura y crear un *quadtree* a partir del mismo. Las partes del escenario donde exista más detalle (en el mapa de altura, habrá muchas alturas diferentes), se subdividirán más veces.

Además al representar la escena, es posible utilizar un heurístico basado en la distancia hasta la cámara y en el detalle del terreno para recorrer el árbol. Al hacerlo de este modo, será posible determinar qué partes hay que pintar con más detalle, seleccionando la representación más simple para los objetos más distantes y la más compleja para los más cercanos y grandes.

En el momento de escribir esta documentación, *InfiniteCode* permite descargar de su web [47] un ejemplo de *Quadtrees* (apoyados en mapas de altura).

Árboles binarios de triángulos (BTT)

Un BBT (Balanced Binary Tree) es un caso especial de un árbol binario. En cada subdivisión se divide al espacio en dos triángulos. El hecho de que cada nodo tenga menos descendientes que un *quadtree* e igualmente menos vecinos, hace que esta estructura sea mejor para algoritmo de nivel de detalle continuo.

Dallaire [23] explica con detalle en *Gamasutra* cómo generar este tipo de estructuras para indexar fragmentos de terreno.

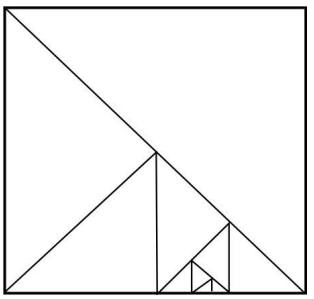


Figura 26.51: División de una superficie con un BTT.

26.7.3. Determinación de la resolución

En una escena de exteriores, a parte de utilizar la estructura de datos correcta, es crucial disponer de un método para determinar la resolución de cada uno de los objetos de la escena.

Cada objeto podrá aparecer con diferentes resoluciones y para ello serán necesarias dos cosas: la forma de seleccionar la resolución correcta y la manera de representarla, esto es, un algoritmo de renderizado que permita hacerlo de manera correcta.

Determinar la resolución de manera exacta no es un problema abordable en un tiempo razonable, puesto que son muchas las variables que influyen en dicha acción. Lo mejor es afrontar este problema utilizando algún heurístico permita aproximar una buena solución.

Un primer enfoque podría ser utilizar la distancia desde la cámara hasta el objeto, y cambiar la resolución del mismo según objeto se acerca (más resolución) o se aleja (menos resolución). Puede que haya un objeto muy lejano pero que sea tan grande que requiera un poco más del detalle que le corresponda según la distancia. Un heurístico añadido que se podría utilizar es el número de píxeles que ocupa aproximadamente en el espacio de pantalla, utilizando una *bouding-box* y proyectándola sobre la misma. Incluso se podría utilizar el hardware, como se ha visto en el tema anterior, para realizar estas comprobaciones.

Una vez que se ha determinado cómo se selecciona la resolución el siguiente paso será aplicar la política de dibujado. Existe una gran división, dependiendo de la continuidad de los modelos a pintar. Si se almacenan diferentes modelos de un mismo objeto, con niveles diferentes de detalle, hablaremos de una política discreta de LOD. Si el detalle de los modelos se calcula en tiempo real según el criterio de resolución, hablaremos de una política de LOD continua.

Políticas Discretas de LOD

Si se utiliza una política discreta, se tendrán varias representaciones del mismo objeto, con diferentes nivel de detalle. Este nivel de detalle irá desde la versión original, con el mayor detalle posible, a una versión con muy pocos triángulos. Se creará la versión de alto detalle y se generarán versiones simplificadas reduciendo el número de triángulos gradualmente con alguna herramienta de diseño 3D.

Teniendo una tabla con diferentes modelos donde elegir, el algoritmo de pintado simplemente tiene que elegir el que corresponda y ponerlo en la cola de renderizado. El problema de estas políticas es que existen un momento en el que se produce un cambio notable en el objeto, y es completamente perceptible si no se disimula de alguna forma.

Una de las técnica que se usa para ocultar el salto que se produce al cambiar de modelo es utilizar *alpha blending* entre el modelo origen y destino. Este efecto se puede ver como un *cross-fade* entre los mismos, cuya intensidad dependerá del heurístico utilizado. Así en la mitad de la transición, cada modelo se renderizará con un *alpha* de 0.5 (o del 50 %). Justo antes de empezar la transición, el modelo origen tendrá un valor *alpha* de 1 y el destino de 0, y al finalizar tendrán los valores intercambiados. Un inconveniente muy importante de esta técnica es que durante un tiempo durante el cual antes sólo se representaba un objeto, ahora se representarán dos, lo que supone una sobrecarga de la GPU.

Políticas Continuas de LOD

Si se quiere evitar del todo el salto producido por el intercambio de modelos, se podría implementar una forma de reducir el número de triángulos en tiempo real, y generar un modelo dependiendo de la resolución requerida.

Este tipo de cálculos en tiempo real son muy costosos, porque hay que determinar qué aristas, vértices o triángulos se pueden eliminar y además aplicar esa modificación al modelo.

Hoppe [44] propone una implementación eficiente de lo que llama mallas progresivas (*Progressive Meshes*). La técnica se basa en la eliminación de aristas de la malla, convirtiendo dos triángulos en sólo uno por cada arista eliminada (*edge-collapsing*). Hoppe determina que esta técnica es suficiente para simplificar mallas y propone algunos heurísticos para eliminar las aristas.

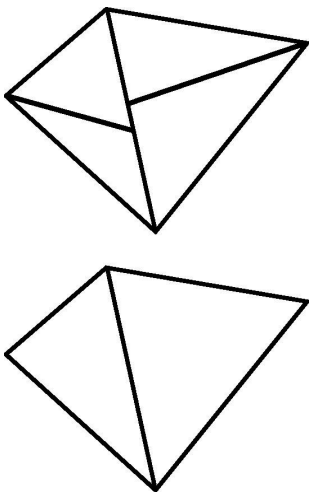


Figura 26.52: *Edge-Collapsing*. Arriba, el *strip* original. Abajo, el nuevo *strip* después de aplicar la técnica.

Hay dos posibles aproximaciones a la hora de quitar una arista, la primera, crear un nuevo vértice en el centro de la misma, y la otra eliminarla completamente (más eficiente). La primera aproximación es válida para todas las aristas, la segunda es sólo válida para aristas que corten a triángulos y no a cuadriláteros, puesto que al eliminarla se debería obtener un polígono con tres vértices.

Un posible heurístico para utilizar es el ángulo que forman los dos triángulos conectados por dicha arista. Si el ángulo es menor que un umbral, se quitará esa arista. Como optimización, esta técnica no debería utilizarse en cada *frame*, sino sólo cuando cambie la distancia o el área que ocupa el objeto en pantalla lo suficiente.

Este tipo de políticas permite obtener el mejor resultado, a costa de añadir un coste computacional. Además, otra desventaja muy importante es que la información de mapeado de las texturas del objeto se podrá ver afectada por la reducción del número de triángulos.

26.7.4. Técnicas y Algoritmos

Las estructuras de datos presentadas anteriormente se utilizan en las técnicas y algoritmos que se presentan a continuación.

GeoMipmapping

De Boer [25] presenta el *GeoMipmapping* como una técnica para representar de manera eficiente grandes terrenos. En su artículo, De Boer divide el algoritmo en tres fases diferentes: la representación en memoria de los datos del terreno y a qué corresponderá en la representación, el *frustum culling* con los pedazos de terreno disponibles (*chunks*) y por último, describe los *GeoMipMaps* haciendo una analogía con la técnica de *mipmapping* usada para la generación de texturas.

Representación del terreno

La representación del terreno elegida es la de una malla de triángulos cuyos vértices están separados por la misma distancia en el eje X y en el eje Z.

El número de vértices horizontales y verticales de la malla tendrá que ser de la forma $2^n + 1$, lo que significa tener mallas con 2^n cuadriláteros, que tendrán 4 vértices compartidos con sus vecinos. Cada cuadrilátero está compuesto de dos triángulos, que serán los que se mandarán a la cola de representación.

Cada vértice tendrá un valor fijo de X y de Z, que no cambiará durante el desarrollo del algoritmo. El valor de Y (altura) será leído de un mapa de altura de *8-bit*, que tendrá exactamente las misma dimensiones que la malla. Posteriormente esta se cortará en pedazos de tamaño $2^n + 1$. Estos pedazos se usarán en un *quadtree* para realizar el *frustum culling*, y como primitivas de nivel 0 para los *GeoMipMaps*. En la figura 26.53 se muestra una de estas mallas, donde n vale 2.

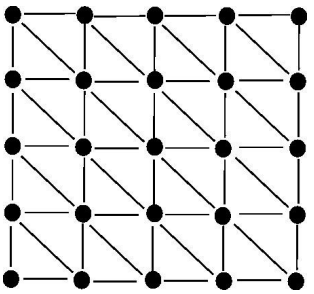


Figura 26.53: Malla con los datos del terreno. Cada círculo es un vértice.

Una ventaja de utilizar este tipo de representación es que los pedazos de malla se pueden mandar como una sola primitiva (*strips*) el hardware. La desventaja es que los vértices de los 4 bordes de dichos trozos se comparten con los bloques que lo rodean, y se transformarán dos veces.

View-Frustum Culling

Será necesario descartar los pedazos de terreno que no caigan dentro del *frustum* de la vista (cámara) puesto que no serán visibles. Para ello, lo ideal es utilizar un *quadtree*.

El *quadtree* será precalculado antes de comenzar la parte interactiva de la aplicación y consistirá tan solo en *bounding boxes* de tres dimensiones, que a su vez contendrán otras correspondientes a los subnodos del nodo padre. En cada hoja del árbol quedará un pedazo del nivel 0 de divisiones que se ha visto al principio. Es suficiente utilizar *quadtrees* y no *octrees* ya que la división se realiza de la superficie, y no del espacio.

Para descartar partes del terreno, se recorrerá el árbol desde la raíz, comprobando si la *bounding box* está dentro del *frustum* al menos parcialmente, y marcando dicho nodo en caso afirmativo. Si una hoja está marcada, quiere decir que será visible y que se mandará a la cola de representación. A no ser que el terreno sea muy pequeño, se terminarán mandando muchos triángulos a la cola, y esta optimización no será suficiente. Es aquí donde De Boer introduce el concepto de *Geomipmapping*.

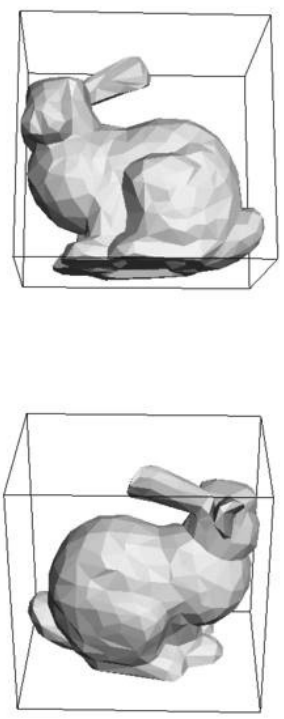


Figura 26.54: Ejemplo de una *bounding-box* (MATH IMAGES PROJECT).

Geomipmaps y nivel de detalle

Esta técnica se basa en el hecho de que los bloques que están más lejos de la cámara no necesitan representarse con tan nivel de detalle como los más cercanos. De este modo, podrán ser representados con un número mucho menor de triángulos, lo que reducirá enormemente el número de triángulos del terreno que se mandarán a la cola de renderizado. Otro algorítmos utilizan una aproximación en la que hay que analizar cada triángulo para poder aplicar una política de nivel de detalle. Al contrario, esta técnica propone una política discreta que se aplicará en un nivel más alto.

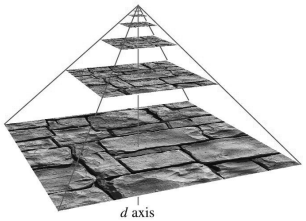


Figura 26.55: Construcción de los *mipmaps*. En cada nivel, la textura se reduce a un cuarto de su área. (AKENINE-MOLLER)

La técnica clásica de *mipmapping* se aplica a las texturas, y consiste en la generación de varios niveles de subtexturas a partir de la original. Este conjunto de texturas se utilizan en una política de nivel de detalle para texturas, usando unas u otras dependiendo de la distancia a la cámara. Esta es la idea que se va a aplicar a la mallas 3D de terrenos.

Cada bloque de terreno tendrá asociado varios *mipmaps*, donde el original corresponde al bloque del mapa de altura. Estos *GeoMipMaps* pueden ser precalculados y almacenados en memoria para poder ser utilizados en tiempo de ejecución directamente.

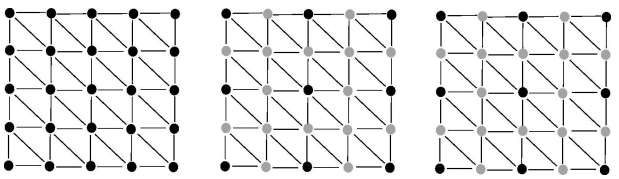


Figura 26.56: Diferentes niveles de detalle de la malla desde los *GeoMipMaps*: niveles 0, 1 y 2.

Para elegir qué *geomipmap* es adecuado para cada distancia y evitar saltos (producto de usar distancias fijas para cada uno) habrá que utilizar un método un poco más elaborado. En el momento en que se se pase del nivel 0 al 1 (ver figura 26.56) existirá un error en la representación del terreno, que vendrá dado por la diferencia en altura entre ambas representaciones. Debido a la perspectiva, la percepción del error tendrá que ver con los píxeles en pantalla a la que corresponda esa diferencia. Ya que cada nivel tiene muchos cambios en altura, se utilizará el máximo, que podrá almacenarse durante la generación para realizar decisiones más rápidas. Si el error en píxeles cometido es menor que un umbral, se utilizará un nivel más elevado.

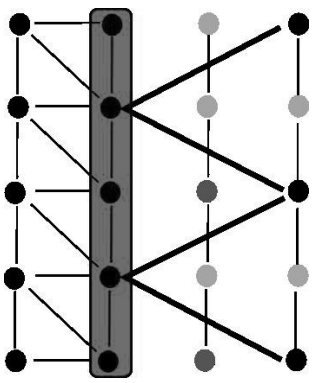


Figura 26.57: Unión de diferentes niveles de *GeoMipmaps*. En rojo la frontera común.

Hay que tener en cuenta que la mayoría de las ocasiones el terreno estará formado por bloques con deferentes niveles, lo que puede hacer que existan vértices no conectados. Será necesario reorganizar las conexiones entre los mismos, creando nuevas aristas.

En la figura 26.57 se muestra una propuesta que consiste en conectar los vértices de la malla de nivel superior con los de la de nivel inferior pero saltando un vértice cada vez.

ROAM

Duchaineau [29] propone ROAM (Real-time Optimally Adapting Meshes) como un enfoque de nivel de detalle continuo al problema de la representación de exteriores. El algoritmo que propone combina una buena representación del terreno (lo que facilitará el *culling*) con un nivel de detalle dinámico que cambiará la resolución del terreno según la disposición de la cámara. ROAM es un algoritmo complejo, como el de los BSPs, que está dividido en dos pasadas y que permite una representación muy rápida de terrenos.

En este algoritmo, la malla se calcula en tiempo real, utilizando precálculos sobre la resolución necesaria en cada caso. En la primera pasada se rellena un BTT con la información geográfica, añadiendo información sobre el error producido al entrar en un subnodo (que será usada para detectar zonas que necesiten un mayor nivel de detalle). En la segunda pasada se construirá otro BTT, que será el encargado de crear la maya y de representar el terreno.

Primera pasada: Árbol de Varianza.

En la primera pasada se construirá un árbol que almacenará el nivel de detalle que existe en el terreno. Una buena métrica es la varianza. En cada hoja del árbol (un píxel de un mapa de altura), la varianza almacenada será por ejemplo la media de los píxeles que lo rodean. La varianza de los nodos superiores será la máxima varianza de los nodos hijos.

Segunda pasada: Malla.

Se utilizará un BTT, donde el nodo raíz representa un terreno triangular (si se quiere representar un terreno rectangular se necesitarán los nodos como este). Se almacenará la información de la altura para cada una de las esquinas del triángulo almacenado en cada nodo.

Si entre los vértices de este triángulo grande la información del terreno no es coplanar, se consultará el árbol de varianza para determinar si es conveniente explorar una nueva división para añadir un nivel más de detalle. Se rellenará el BTT hasta que no queden píxeles por añadir del mapa de altura, o hasta que el estadístico sea menor que un umbral elegido.

De este modo se construye un árbol que representa la malla, donde cada nivel del mismo corresponde a un conjunto de triángulos que pueden ser encolados para su renderizado. El árbol podrá expandirse en tiempo real si fuera necesario. Bajar un nivel en el árbol equivale a una operación de partición y subir a una de unión.

El problema viene de la unión de regiones triangulares con diferente nivel de detalle, ya que si no se tienen en cuenta aparecerán agujeros en la malla representada. Mientras que en el *geomipmapping* se parchea la malla, en ROAM, cuando se detecta una discontinuidad se utiliza un *oversampling* de los bloques vecinos para asegurar que están conectados de manera correcta. Para ello, se añade información a cada lado de un triángulo y se utilizan alguna reglas que garantizarán la continuidad de la malla.

Se conoce como vecino base de un triángulo al que está conectado a este a través de la hipotenusa. A los otros dos triángulos vecinos se los conocerá como vecino izquierdo y derecho (figura 26.59). Analizando diferentes árboles se deduce que el vecino base será del mismo nivel o del anterior (menos fino) nivel de detalle, mientras que los otros vecinos podrán ser del mismo nivel o de uno más fino.

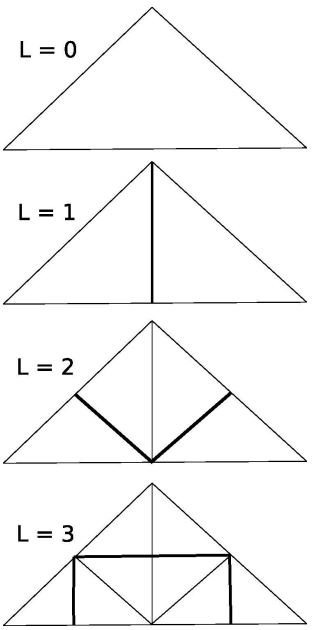


Figura 26.58: Distintos niveles de división en un BTT.

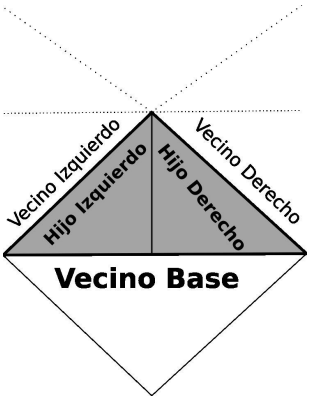


Figura 26.59: Etiquetado de un triángulo en ROAM. El triángulo junto a su vecino base forman un diamante.

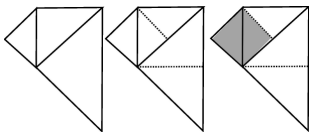


Figura 26.60: Partición recursiva hasta encontrar un diamante para el nodo de la izquierda.

Las reglas propuestas para seguir explorando el árbol y evitar roturas en la malla serán las siguientes:

- Si un nodo es parte de un diamante, partir el nodo y el vecino base.
- Si se está en el borde de una malla, partir el nodo.
- Si no forma parte de un diamante, partir el nodo vecino de forma recursiva hasta encontrar uno antes de partir el nodo actual (figura 26.60).

El algoritmo en tiempo de ejecución recorrerá el árbol utilizando alguna métrica para determinar cuándo tiene que profundizar en la jerarquía, y cómo ha de realizar las particiones cuando lo haga dependerá de las reglas anteriores.

Realizar en tiempo real todos estos cálculos es muy costoso. Para reducir este coste, se puede dividir el terreno en *arrays* de BTTs y sólo recalcular el árbol cada ciertos *frames* y cuando la cámara se haya movido lo suficiente para que cambie la vista por encima de un umbral.

Otra aproximación para optimizar el algoritmo es utilizar el *frustum* para determinar qué nodos hay que reconstruir de nuevo, marcando los triángulos que quedan dentro o fuera, o parcialmente dentro. Sólo estos últimos necesitarán atención para mantener el árbol actualizado. A la hora de representarlo, lo único que habrá que hacer es mandar los nodos marcados como dentro, total o parcialmente.

Chunked LODs

Ulrich [100] propone un método para representar grandes extensiones de tierra. En la demo del SIGGRAPH 2002 incluye un terreno que cubre $160Km^2$. Este método tiene su partida en una imagen muy grande, por ejemplo obtenida de un satélite, lo que hace el método ideal para la implementación de simuladores de vuelo.

Para almacenar la información se utilizará un *quadtree*. Se comenzará con una imagen potencia de dos en el nodo raíz, que corresponderá a una imagen de muy baja resolución del terreno completo. Según se vaya profundizando, los 4 subnodos hijos contendrán imágenes del mismo tamaño pero con la calidad resultante de hacer un zoom a los cuatro cuadrantes de la misma (figura 26.61).

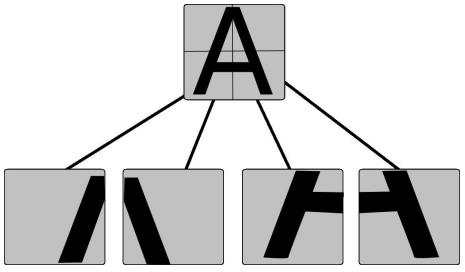


Figura 26.61: Nodo y subnodos de las texturas en un *quadtree* para Chunked LODs.

A cada uno de los nodos se le añade información acerca de la pérdida del nivel de detalle se produce al subir un nivel en la jerarquía. Si las hojas contienen imágenes de 32x32 píxeles, los pedazos de terreno contendrán 32x32 vértices. El árbol de mallas también forma parte del preproceso normalmente, partiendo de una malla muy grande y con mucha resolución y partiéndola en trozos más pequeños.

Para renderizar el *quadtree* bastará con recorrerlo realizando *clipping* jerárquico, utilizando algún umbral para comparar con un valor obtenido de computar la distancia a la cámara, el error que produce un nivel determinado y la proyección de perspectiva que se está utilizando. Si la comparación sugiere que el nivel de detalle debería incrementarse en esa región, se procederá a encolar un nivel más bajo del árbol recursivamente.

Ulrich propone unir los *chunks* simplemente prolongando un poco sus fronteras con unos faldones. Como la proyección de la textura dependerá sólo de la X y la Z, las uniones serán prácticamente invisibles. Este algoritmo se beneficia de un *quadtree* para texturizar el terreno. Como contrapartida, el principal problema del mismo es la gran cantidad de memoria que necesita (la demo del SIGGRAPH, más de 4GB), lo que hace que sea necesario prestar especial atención a la misma, cargando en memoria los nuevos pedazos cuando sean necesarios, descartando los que no se necesitan hace tiempo.

Terrenos y GPU

En su libro, Dalmau propone una aproximación diferente a la representación de terrenos utilizando simplemente la GPU. La premisa de la que parte el algoritmo es mantener a la CPU completamente desocupada, pudiendo ser utilizada esta para otra labores como para la inteligencia artificial o para el cálculo de las colisiones o las físicas.

De esto modo sugiere que la geometría del terreno tendrá que almacenarse en algún modo en el que la GPU pueda acceder a ella sin la intervención de la CPU, seleccionando bloques de 17x17 vértices (512 triángulos), que serán analizados e indexados para maximizar el rendimiento. Además, como diferentes bloques compartirán vértices, estos se almacenarán sólo una vez y se indexarán de forma eficiente. Así, la CPU sólo tendrá que determinar los bloques visibles y mandar esta información a la GPU para que los pinte.

A esta técnica se le puede sumar el uso de *bounding boxes* para cada bloque de terreno y la construcción de un PVS o incluso implementar alguna política de LOD (que Dalmau define como no necesaria excepto en GPU con limitaciones muy restrictivas de número de triángulos por segundo).

Scenegraphs de Exteriores

Una escena de exteriores es mucho más grande que una de interiores. La cantidad de datos que es necesario manejar es en consecuencia gigantesca en comparaciones con escenas mucho más pequeñas. Aunque se tenga una buena política de nivel de detalle, hay que tener en cuenta que el número de triángulos totales que se tendrán que manejar es enorme.

Para implementar un grafo de escena de exteriores es importante tener en cuenta que:

- Cada objeto sólo tendrá una instancia, y lo único que se almacenará aparte de esta serán enlaces a la misma.
- Es necesario implementar alguna política de nivel de detalle, puesto que es imposible mostrar por pantalla (e incluso mantener en memoria) absolutamente todos los triángulos que se ven en una escena.
- Se necesita una rutina muy rápida para descartar porciones no visibles del terreno y del resto de objetos de la escena.

Mientras que para almacenar un terreno y las capas estáticas que lleva encima es muy buena opción utilizar un *quadtree*, será mejor utilizar alguna tabla de tipo rejilla para almacenar la posición de los objetos dinámicos para determinar cuáles de ellos se pintar en cada fotograma de manera rápida.

¿Reconoce alguna de estas técnicas en algún juego o aplicación que haya utilizado recientemente?

26.7.5. Exteriores y LOD en OGRE

OGRE da soporte a diferentes estilos de escenas. El único *scene manager* que es de uso exclusivo para interiores es `ST_INTERIOR` (el gestor de portales parece estar abandonado en los ejemplos de ogre 1.7.3). El resto gestores está más o menos preparado para escenas de exteriores. Estos son:

- **`ST_GENERIC`** - Gestor de propósito general, adecuado para todo tipo de escenas, pero poco especializado. Se utiliza un *octree* para almacenar los objetos.
- **`ST_EXTERIOR_CLOSE`** - Gestor de terrenos antiguo de OGRE. Soportado hasta la versión 1.7, deprecado en la 1.8.
- **`ST_EXTERIOR_REAL_FAR`** - Gestor que permite dividir la escena en un conjunto de páginas. Sólo se cargarán las páginas que se usen en un momento determinado, permitiendo representar escenas muy grandes de cualquier tamaño. Cada página tiene asociado un mapa de altura, y se pueden aplicar diferentes texturas a la malla generada según la altura.

Terrenos

OGRE soporta la creación de terrenos utilizando mapas de altura, que se pueden cargar en diferentes páginas y cubrir grandes extensiones en una escena.

En estos momentos OGRE está cambiando de gestor de terrenos. El gestor nuevo utiliza *shaders* que dependen del plugin *CgProgramManager*, válido únicamente para tarjetas gráficas NVIDIA. Esta dependencia se da sólo utilizando en sistema de render basado en OpenGL, porque lo mientras en las demos de OGRE de Windows funciona perfectamente con DirectX, en GNU/Linux hará falta cargar este *plugin*. En Debian, ya no se distribuye este plugin con lo que será necesario compilarlo desde cero.

Un terreno de OGRE implementa una política de nivel de detalle continua. El nuevo gestor implementa una variante de *Chunked-LODS* y permite cargar mapas muy grandes, compuestos por diferentes mapas de altura.

CG Plugin

Para compilar el plugin *CgManager* de OGRE en Debian, habrá que instalar antes *nvidia-cg-toolkit*, que es privativo.

A continuación se muestra un ejemplo de uso del nuevo terreno de OGRE. El *Scene Manager* que utiliza es el genérico. El mapa de altura que cargará por secciones en el mismo.

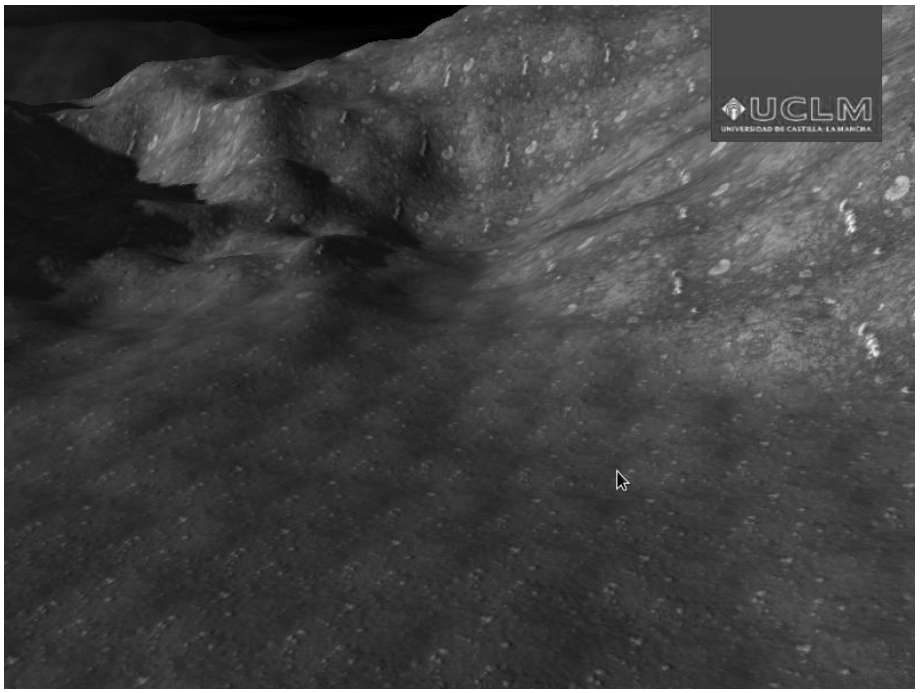


Figura 26.62: Pantallazo de terreno en OGRE

En la lineas [3-5] se configura niebla en la escena, usando un color oscuro para hacer que parezca oscuridad. En las líneas [7-14] se configura una luz direccional que se utilizará en el terreno. A partir de ella se calcularán las sombras en el mismo. En la línea [16] se configura la luz ambiental de la escena. En la [19] se crea un nuevo objeto de configuración del terreno, que se rellenará más adelante. En la [20] se crea un objeto agrupador de terrenos. Este objeto es el responsable de agrupar diferentes pedazos del terreno juntos para que puedan ser representados como proceda. En la línea [26] se configura el error máximo medido en píxeles de pantalla que se permitirá al representar el terreno. En las líneas [28-31] se configura la textura que compodrá con él, dibujando las sombras. Primero se determina la distancia de representación de las luces, luego se selecciona la dirección de las sombras, que parte de una luz direccional y luego se configuran el valor ambiental y difuso de iluminación.

Listado 26.40: Ejemplo de terreno en OGRE. Basado en OGRE3D WIKI

```
1  void MyApp::createScene()
2  {
3    _sMgr->setFog(Ogre::FOG_LINEAR,
4                  Ogre::ColourValue(0.1, 0.1, 0.1),
5                  0.5, 2000, 5000);
6
7    Ogre::Vector3 lightdir(0.55, -0.3, 0.75);
8    lightdir.normalise();
9
10   Ogre::Light* light = _sMgr->createLight("DirLight");
11   light->setType(Ogre::Light::LT_DIRECTIONAL);
12   light->setDirection(lightdir);
13   light->setDiffuseColour(Ogre::ColourValue::White);
14   light->setSpecularColour(Ogre::ColourValue(0.4, 0.4, 0.4));
15
16   _sMgr->setAmbientLight(Ogre::ColourValue(0.2, 0.2, 0.2));
```

```
17
18
19   _tGlobals = OGRE_NEW Ogre::TerrainGlobalOptions();
20   _tGroup = OGRE_NEW Ogre::TerrainGroup(_sMgr,
21                                         Ogre::Terrain::ALIGN_X_Z,
22                                         513, 12000.0f);
23
24   _tGroup->setOrigin(Ogre::Vector3::ZERO);
25
26   _tGlobals->setMaxPixelError(8);
27
28   _tGlobals->setCompositeMapDistance(3000);
29   _tGlobals->setLightMapDirection(light->getDerivedDirection());
30   _tGlobals->setCompositeMapAmbient(_sMgr->getAmbientLight());
31   _tGlobals->setCompositeMapDiffuse(light->getDiffuseColour());
32
33   Ogre::Terrain::ImportData& di;
34   di = _tGroup->getDefaultImportSettings();
35
36   di.terrainSize  = 513;
37   di.worldSize    = 12000.0f;
38   di.inputScale   = 600;
39   di.minBatchSize = 33;
40   di.maxBatchSize = 65;
41
42   di.layerList.resize(3);
43   di.layerList[0].worldSize = 100;
44   di.layerList[0].textureNames.push_back("dirt_diff_spec.png");
45   di.layerList[0].textureNames.push_back("dirt_normal.png");
46   di.layerList[1].worldSize = 30;
47   di.layerList[1].textureNames.push_back("grass_diff_spec.png");
48   di.layerList[1].textureNames.push_back("grass_normal.png");
49   di.layerList[2].worldSize = 200;
50   di.layerList[2].textureNames.push_back("growth_diff_spec.png");
51   di.layerList[2].textureNames.push_back("growth_normal.png");
52
53
54   Ogre::Image im;
55   im.load("terrain.png",
56           Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
57
58   long x = 0;
59   long y = 0;
60
61   Ogre::String filename = _tGroup->generateFilename(x, y);
62   if (Ogre::ResourceGroupManager::getSingleton().resourceExists(
         _tGroup->getResourceGroup(), filename))
63   {
64     _tGroup->defineTerrain(x, y);
65   }
66   else
67   {
68     _tGroup->defineTerrain(x, y, &im);
69     _tsImported = true;
70   }
71
72   _tGroup->loadAllTerrains(true);
73
74   if (_tsImported)
75   {
76     Ogre::TerrainGroup::TerrainIterator ti;
77     ti = _tGroup->getTerrainIterator();
78     while(ti.hasMoreElements())
79     {
80       Ogre::Terrain* t = ti.getNext()->instance;
81       initBlendMaps(t);
82     }
83   }
84
85   _tGroup->freeTemporaryResources();
86
```

```
 87   Ogre::Plane plane;
 88   plane.d = 100;
 89   plane.normal = Ogre::Vector3::NEGATIVE_UNIT_Y;
 90
 91   _sMgr->setSkyPlane(true, plane,"Skybox/SpaceSkyPlane",
 92                      500, 20, true, 0.5, 150, 150);
 93
 94 }
 95
 96 void MyApp::initBlendMaps(Ogre::Terrain* t)
 97 {
 98   Ogre::TerrainLayerBlendMap* blendMap0 = t->getLayerBlendMap(1);
 99   Ogre::TerrainLayerBlendMap* blendMap1 = t->getLayerBlendMap(2);
100
101   Ogre::Real minHeight0  = 70;
102   Ogre::Real fadeDist0   = 40;
103   Ogre::Real minHeight1  = 70;
104   Ogre::Real fadeDist1   = 15;
105
106   float* pBlend1 = blendMap1->getBlendPointer();
107   for (Ogre::uint16 y = 0; y < t->getLayerBlendMapSize(); ++y) {
108     for (Ogre::uint16 x = 0; x < t->getLayerBlendMapSize(); ++x){
109       Ogre::Real tx, ty;
110
111       blendMap0->convertImageToTerrainSpace(x, y, &tx, &ty);
112       Ogre::Real height = t->getHeightAtTerrainPosition(tx, ty);
113       Ogre::Real val = (height - minHeight0) / fadeDist0;
114       val = Ogre::Math::Clamp(val, (Ogre::Real)0, (Ogre::Real)1);
115       val = (height - minHeight1) / fadeDist1;
116       val = Ogre::Math::Clamp(val, (Ogre::Real)0, (Ogre::Real)1);
117       *pBlend1++ = val;
118     }
119   }
120   blendMap0->dirty();
121   blendMap1->dirty();
122   blendMap0->update();
123   blendMap1->update();
124 }
```

En (34) se obtiene la instancia que configura algunos parámetros del terreno. El primero es el tamaño del terreno, que corresponde al número de píxeles del mapa de altura. El segundo el tamaño total del mundo, en unidades virtuales. El tercero (`inputScale`) corresponde a la escala aplicada al valor del píxel, que se transformará en la altura del mapa. Las dos siguientes corresponden al tamaño de los bloques de terrenos que se incluirán en la jerarquía (diferentes LOD). Estos tres últimos valores tendrán que ser del tipo $2^n + 1$. El atributo `layerList` contiene un vector de capas, que se rellenará con las texturas (color y mapa de normales en este caso). El atributo `worldSize` corresponde a la relación de tamaño entre la textura y el mundo. En (55) se carga la imagen con el mapa de altura.

Las siguientes líneas son las que asocian el mapa de altura con el terreno que se va a generar. En este ejemplo sólo se genera el subterreno (0,0) con lo que sólo se cargará una imagen. En las siguientes líneas (61-70) se define el terreno, en este caso, sólo para el *slot* (0,0), aunque el ejemplo está preparado para ser extendido fácilmente y añadir la definición algunos más. La llamada `defineTerrain()` expresa la intención de crear ese pedazo de terreno con la imagen que contiene el mapa de altura. La ejecución de este deseo se realiza en la linea (72). Sin esta llamada, el ejemplo no funcionaría puesto que se ejecutaría el siguiente paso sin haber cargado (generado) el terreno.

En el bloque 74-83 se crea la capa de *blending*, esto es, la fusión entre las tres texturas del terreno (habrá dos mapas de *blending*). Esta operación se realiza en el método `initBlendMaps` 96-124. Dependiendo de la altura a la que corresponda un píxel de las imágenes a fusionar (el tamaño ha de ser coherente con el mapa de altura) así será el valor de *blending* aplicado, con lo que se conseguirá que cada altura presente una textura diferente.

En las líneas 87-89 se añade un plano, y justo debajo, en la 91 se usa como un *Skyplane*.

Skyboxes, skydomes y *skyplanes*

Una de las características de una escena de exteriores es que muy probablemente se llegue a ver el horizonte, una gran parte del cielo, o si estamos en un entorno marino o espacial, el abismo o las estrellas. Representar con alto de nivel de detalle los objetos del horizonte es practicamente imposible. Una de las razones es que probablemente alguna de las políticas de *culling* o de nivel de detalle hará que no se pinte ningún objeto a partir de una distancia determinada de la cámara.

La solución adopotada por muchos desarrolladores y que está disponible en OGRE es utilizar imágenes estáticas para representar lo que se ve en el horizonte. En OGRE existen tres formas de representar el entorno que contiene a todos los objetos de la escena. Es decir, no habrá ningún objeto que quede fuera de las imágenes que representan el horizonte. Los objetos que brinda OGRE son de tipo:

- **`skybox`** - Como su nombre indica es una caja. Normalmente son cubos situados a una distancia fija de la cámara. Así, será necesario proporcionar seis texturas, una para cada uno de sus caras.
- **`skydome`** - Corresponde con una cúpula de distancia fija a la cámara. Con una textura es suficiente. Como contrapartida, una *skydome* sólo cubre la mitad de una esfera sin distorsionarse y, mientras una *skybox* posee un suelo, una *skydome* no.
- **`skyplane`** - El cielo está representado por un plano fijo a la cámara, con lo que parecerá infinito. Con una sóla textura es suficiente.

Ejemplo de *Skybox*

Para utitlizar una *skybox* primero es necesario definir un material:

```
material SkyBoxCEDV
{
  technique
  {
    pass
    {
      lighting off
      depth_write off

      texture_unit
      {
        cubic_texture cubemap_fr.jpg cubemap_bk.jpg cubemap_lf.jpg\
           cubemap_rt.jpg  cubemap_up.jpg cubemap_dn.jpg separateUV
        tex_address_mode clamp
      }
    }
```

```
    }
}
```

En este material se deshabilita la escritura en el *buffer* de profundidad (a través de `depth_write off`) para que la representación del cubo no afecte a la visibilidad del resto de los objetos. También se ha de representar sin iluminación (`lighting off`). Con `cubic_texture` se determina que la textura será cúbica. Existen dos formas de declarar qué imágenes se usarán para la misma. La del ejemplo anterior consiste en enumerar las seis caras. El orden es:

`setSkyBox()`

Esta llamada acepta más parámetros, si fuera necesario definir una orientación tendría que usarse un quinto parámetro (el cuarto es un booleano que indica que la caja se dibuja antes que el resto de la escena.

```
<frontal> <posterior> <izquierda> <derecha> <arriba> <abajo>
```

nombre de los archivos que queda delate de "_". En el caso anterior, sería `cubemap.jpg`. El último parámetro puede ser `combinedUVW`, si en una misma imagen están contenidas las seis caras, o `separateUV` si se separan por imágenes. La primera forma utilizará coordenadas de textura 3D, la segunda usará coordenadas 2D ajustando cada imagen a cada una de las caras. Usando `tex_address_mode clamp` hará que los valores de coordenadas de texturizado mayores que 1,0 se queden como 1,0.

Para utilizar una *Skybox* simplemente habrá que ejecutar esta línea:

```
1 sceneManager->setSkyBox(true, "SkyBoxCEDV", 5000);
```

Como cabría esperar, la *skybox* se configura para el gestor de escenas. Ésta se encontrará a una distancia fija de la cámara (el tercer parámetro), y podrá estar activada o no (primer parámetro).

Ejemplo de *SkyDome*

Aunque sólo se utiliza una textura, realmente una *SkyDome* está formada por las cinco caras superiores de un cubo. La diferencia con una *skybox* es la manera en la que se proyecta la textura sobre dichas caras. Las coordenadas de texturizado se generan de forma curvada y por eso se consigue tal efecto de cúpula. Este tipo de objetos es adecuado cuando se necesita un cielo más o menos realista y la escena no va a contener niebla. Funcionan bien con texturas repetitivas como las de nubes. Una curvatura ligera aportará riqueza a una escena grande y una muy pronunciada será más adecuada para una escena más pequeña donde sólo se vea pedazos de cielo de manera intermitente (y el efecto exagerado resulte atractivo).

Un ejemplo de material es el siguiente:

```
material SkyDomeCEDV
{
[...]
      texture_unit
      {
        texture clouds.jpg
        scroll_anim 0.15 0
      }
[...]
}
```

El resto del mismo sería idéntico al ejemplo anterior. Con `scroll_anim` se configura un desplazamiento fijo de la textura, en este caso sólo en el eje X.

La llamada para configurar una *skydome* en nuestra escena en un poco más compleja que para una *skybox*.

```
1 sceneManager->setSkyDome(true, "SkyDomeCEDV", 10, 8, 5000);
```

Donde el tercer parámetro es la curvatura (funciona bien para valores entre 2 y 65), y el cuarto es el número de veces que se repite la textura en la cúpula.

Ejemplo de *SkyPlane*

Un *skyplane* es a priori la más simple de las representaciones. Como su nombre indica está basado en un plano, y la creación de uno es necesaria. Aun así también es posible aplicarle algo de curvatura, siendo más adecuado para escenas con niebla que una cúpula. El material podría ser el mismo que antes, incluso sin utilizar animación. Se ve un ejemplo del uso del mismo en el ejemplo anterior del terreno.

Se propone crear una escena con cada uno de las tres técnicas anteriores. Si fuera posible, utilice el ejemplo de terreno anterior.

LOD : Materiales y Modelos

OGRE da soporte a la representación con diferentes niveles de detalle tanto en materiales como en modelos.

Materiales

En los materiales no sólo se remite a las texturas sino a todas las propiedades editables dentro del bloque `technique`. Para configurar un material con soporte para LOD lo primero es configurar la estrategia (`lod_strategy`) que se va a utilizar de las dos disponibles:

- **`Distance`** - Basado en la distancia desde la cámara hasta la representación del material. Se mide en unidades del mundo.
- **`PixelCount`** - Basado en el número de píxeles del material que se dibujan en la pantalla para esa instancia.

Tras esto, lo siguiente es determinar los valores en los que cambiará el nivel de detalle para dicha estrategia. Se usará para tal labor la palabra reservada `lod_values` seguida de dichos valores. Es importante destacar que tendrán que existir al menos tantos bloques de `technique` como valores, ya que cada uno de ellos estará relacionado con el otro respectivamente. Cada uno de los bloques `technique` debe tener asociado un índice para el nivel de detalle (`lod_index`). Si no aparece, el índice será el 0 que equivale al mayor nivel de detalle, opuestamente a 65535 que corresponde con el menor. Lo normal es sólo tener algunos niveles configurados y probablemente jamás se llegue a una cifra tan grande. Aun así, es importante no dejar grandes saltos y hacer un ajuste más o menos óptimo basado en las pruebas de visualización de la escena. Es posible que varias técnicas tengan el mismo índice de nivel de detalle, siendo OGRE el que elija cuál es la mejor de ellas según el sistema en el que se esté ejecutando.

Un ejemplo de material con varios niveles de detalle:

```
material LOD_CEDV
```

```
{
  lod_values 200 600
  lod_strategy Distance

  technique originalD {
    lod_index 0
    [...]
  }
  technique mediumD {
    lod_index 1
    [...]
  }

  technique lowD {
    lod_index 2
    [...]
  }

  technique shaders {
    lod_index 0
    lod_index 1
    lod_index 2
    [...]
  }
}
```

Nótese que la técnica `shaders` afecta a todos los niveles de detalle.

Como ejercicio se propone construir una escena que contenga 1000 objetos con un material con tres niveles de detalle diferente. Siendo el nivel 0 bastante complejo y el nivel 3 muy simple. Muestre los fotogramas por segundo y justifique el uso de esta técnica. Se sugiere mostrar los objetos desde las diferentes distancias configuradas y habilitar o deshabilitar los niveles de detalle de baja calidad.

Modelos

OGRE proporciona tres formas de utilizar LOD para mallas. La primera es una donde modelos con menos vértices se generan de forma automática, haciendo uso de `progressiveMesh` internamente. Para utilizar la generación automática será necesario que la malla que se cargue no tenga ya de por sí LOD incluido (por ejemplo, la cabeza de ogro de las demos ya contiene esta información y no se podrá aplicar). Una vez que la entidad está creada, se recuperará la instancia de la malla contenida dentro.

¿Podría analizar los niveles de detalle incluidos en la cabeza de ogro de las demos de OGRE? Use `OgreXMLConverter` para obtener una versión legible si es necesario.

Listado 26.41: Generación de niveles de detalle para una malla

```
1  Entity *entidad = sceneManager->createEntity("objeto", "objeto.mesh
       ");
2
3  Ogre::MeshPtr mesh = entidad->getMesh();
4
5  Ogre::ProgressiveMesh::LodValueList lodDvec;
6
7  lodDvec.push_back(50);
8  lodDvec.push_back(100);
9  lodDvec.push_back(150);
10 lodDvec.push_back(200);
11
12 Ogre::ProgressiveMesh::generateLodLevels(mesh, lodDList,
       ProgressiveMesh::VRQ_PROPORTIONAL, 0.1);
```

En la línea [3] del ejemplo anterior se crea una lista de distancias para el nivel de detalle. Esta lista no es más que un vector de reales que determina los valores que se utilizarán según la estrategia elegida para esa entidad. Por defecto corresponde con la distancia y en ese caso los valores corresponden a las raíces cuadradas de la distancia en la que se producen cambios. La estrategia se puede cambiar con `setLodStrategy()` usando como parámetro un objeto de tipo base `lodStrategy`, que será `DistanceLodStrategy` o `PixelCountLodStrategy`. Estas dos clases son *singletons*, y se deberá obtener su instancia utilizando `getSingleton()`. La segunda de ellas corresponde con la política basada en el número de píxeles que se dibujan en la pantalla de una esfera que contiene al objeto (*bounding-sphere*). En la línea [12] se llama a la función que genera los diferentes niveles a partir de la malla original. El segundo parámetro es corresponde con el método que se usará para simplificar la malla:

- **`VRQ_CONSTANT`** - Se elimina un número dijo de vértices cada iteración.
- **`VRQ_PROPORTIONAL`** - Se elimina una cantidad proporcional al número de vértices que queda en la malla.

El tercero corresponde con el valor asociado al método y en el caso de ser un porcentaje tendrá que estar acotado entre 0 y 1.

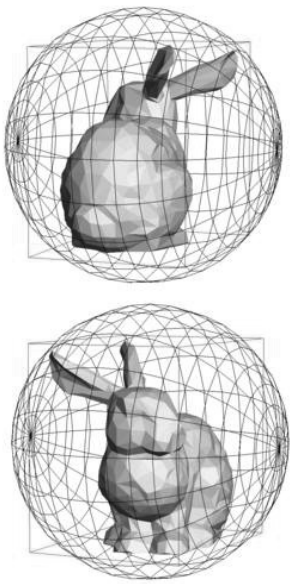


Figura 26.63: Ejemplo de *bounding-sphere*(MATH IMAGES PROJECT)

La segunda forma de añadir LOD a una entidad de OGRE es hacerlo de manera completamente manual. Será necesario crear diferentes mallas para el mismo objeto. Esto se puede llevar a cabo en un programa de edición 3D como Blender. Se empieza con el modelo original, y se va aplicando algún tipo de simplificación, guardando cada uno de los modelos obtenidos, que serán usados como mallas de diferente nivel de detalle para la entidad que represente a dicho modelo.

Listado 26.42: Generación de niveles de detalle para una malla

```
1  Ogre::MeshPtr m =
2    Ogre::MeshManager::getSingleton().load("original.mesh",
3     Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
4
5  m->createManualLodLevel(250, "lod2.mesh");
6  m->createManualLodLevel(500, "lod3.mesh");
7  m->createManualLodLevel(750, "lod3.mesh");
8
9  Ogre::Entity* objeto = sceneManager->createEntity("objetoConLOD",
10                                                   m->getName() );
11
12 Ogre::SceneNode* n = sceneManager->getRootSceneNode()->
       createChildSceneNode();
13 n->attachObject(objeto);
```

La tercera forma es que el archivo que se carga ya tenga la información de nivel de detalle. Esta forma es bastante habitual, ya que el programa `OgreXMLConverter` acepta los siguientes parámetros relacionados con en nivel de detalle cuando se convierte un XML en un archivo *mesh* de OGRE:

- **`-l <lodlevels>`** - El número de niveles de detalle.
- **`-v <lodvalue>`** - El valor asociado a la reducción de detalle.
- **`-s <lodstrategy>`** - La estrategia a usar. Puede ser *Distance* o *PixelCount*.
- **`-p <lodpercent>`** - Porcentaje de triángulos reducidos por iteración.
- **`-f <lodnumtrs>`** - Número fijo de vértices para quitar en cada iteración.

Cuando se cargue un archivo *mesh* creado de este modo, se utilizará de forma automática el número de niveles de detalle configurados.

Se propone la construcción de una escena con unos 1000 objetos utilizando cualquiera de las dos primeras formas programáticas de añadir LOD. Justifique mediante el número de *frames* por segundo el uso de este tipo de técnicas. Compare la velocidad usando y sin usar LOD.

26.7.6. Conclusiones

En los dos últimas secciones se ha realizado una introducción a algunas de las técnicas utilizadas para aumentar el rendimiento de las representaciones en tiempo real. Sin algunas de ellas, sería del todo imposible llevar a cabo las mismas. Si bien es cierto que cualquier motor gráfico actual soluciona este problema, brindando al programador las herramientas necesarias para pasar por alto su complejidad, se debería conocer al menos de forma aproximada en qué consisten las optimizaciones relacionadas.

Saber elegir la técnica correcta es importante, y conocer cuál se está utilizando también. De esto modo, será posible explicar el comportamiento del juego en determinadas escenas y saber qué acciones hay que llevar a cabo para mejorarla.

Por desgracia, OGRE está cambiando alguno de los gestores de escena y se está haciendo evidente en la transición de versión que está sucediendo en el momento de escribir esta documentación.

La forma de realizar optimizaciones de estos tipos ha cambiado con el tiempo, pasando de utilizar el ingenio para reducir las consultas usando hardware genérico, a usar algoritmos fuertemente apoyados en la GPU.

Sea como sea, la evolución de la representación en tiempo real no sólo pasa por esperar a que los fabricantes aceleren sus productos, o que aparezca un nuevo paradigma, sino que requiere de esfuerzos constantes para crear un código óptimo, usando los algoritmos correctos y eligiendo las estructuras de datos más adecuadas para cada caso.

Capítulo 27

Plataformas Móviles

Miguel García Corchero

Motores de videojuego

Existen otros motores de videojuegos más utilizados en la industria como CryEngine o Unreal Engine y tienen más funcionalidades que Unity3D pero también es más complejo trabajar con ellos.

Un **motor de videojuegos** es un termino que hace referencia a una serie de **herramientas** que permiten el diseño, la creación y la representación de un videojuego. La funcionalidad básica de un motor es proveer al videojuego renderización, gestión de físicas, colisiones, scripting, animación, administración de memoria o gestión del sonidos entre otras cosas.

En este capítulo se trata el caso específico del motor de videojuegos **Unity3D** y se realizará un repaso superficial por la forma de trabajar con un motor de videojuegos mientras se realiza un videjuego de ejemplo para dispositivos móviles con el sistema operativo **iOS** o **Android**.

El videojuego será un *shoot'em up* de aviones con vista cenital.

27.1. Método de trabajo con un motor de videojuegos

27.1.1. Generación de contenido externo al motor

- **Diseño del videojuego**: Esta fase suele hacerse con papel y boli. En ella definiremos las mecánicas necesarias a implementar, haremos bocetos de los personajes o situaciones implicadas y creamos listas de tareas a realizar asignando diferentes prioridades.

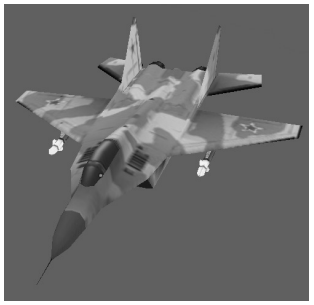


Figura 27.1: Visualización del modelo 3D del jugador.

- **Generación del material gráfico**: El videojuego necesitará gráficos, texturas, fuentes, animaciones o sonidos. Este material se conoce como assets. Podemos utilizar diferentes programas de modelado y texturizado 3D o 2D para esta tarea, ya que todos los assets son exportados a un formato de entrada de los que reconoce el motor gráfico.

27.1.2. Generación de contenido interno al motor

- **Escritura de scripts**: La escritura de scripts se realiza con un editor de texto externo a las herramientas del motor gráfico pero se considera contenido íntimamente relacionado del motor de videojuego.
- **Escritura de shaders**: Los shaders también se escriben con un editor externo.
- **Importación de assets**: Uno de los pasos iniciales dentro del entorno integrado de Unity3D es añadir al proyecto todo el material generado anteriormente y ajustar sus atributos; como formatos, tamaños de textura, ajuste de propiedades, calculo de normales, etc.
- **Creación de escenas**: Crearemos una escena por cada nivel o conjunto de menús del videojuego. En la escena estableceremos relaciones entre objetos y crearemos instancias de ellos.
- **Creación de prefabs**: Los *prefabs* son agrupaciones de objetos que se salvan como un objeto con entidad propia.
- **Optimización de la escena**: Uno de los pasos fundamentales se lleva a cabo al final del desarrollo de la escena y es la optimización. Para ello emplearemos técnicas de lightmapping y occlusion culling con las herramientas del entorno.

Scripts

Los scripts que utilizamos se pueden escribir en los lenguajes C#, Javascript o BOO.

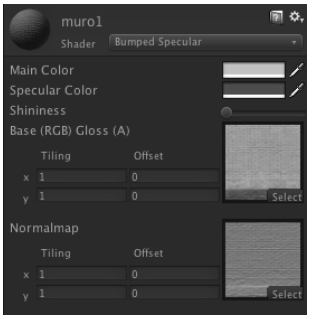


Figura 27.2: Visualización de un shader de normal mapping.

Con Unity3D no tenemos que preocuparnos de la gestión de las físicas, colisiones, renderización o controles a bajo nivel. Nos dedicamos únicamente a la programación de scripts y shaders.

27.2. Creación de escenas

Una escena está constituida por instancias de objetos de nuestros assets y las relaciones entre ellos. Podemos considerar una escena como la serialización de el objeto escena. Este objeto contiene jerárquicamente otros objetos que son almacenados cuando se produce esa serialización, para posteriormente cargar la escena con todas esas instancias que contendrán los mismos valores que cuando fueron almacenados. Hay **dos enfoques** diferentes a la hora de crear escenas:

1. **Una escena por cada nivel del juego**: Utilizaremos este enfoque cuando cada nivel tenga elementos diferentes. Podrán repetirse elementos de otros niveles, pero trataremos que estos elementos sean prefabs para que si los modificamos en alguna escena, el cambio se produzca en todas.

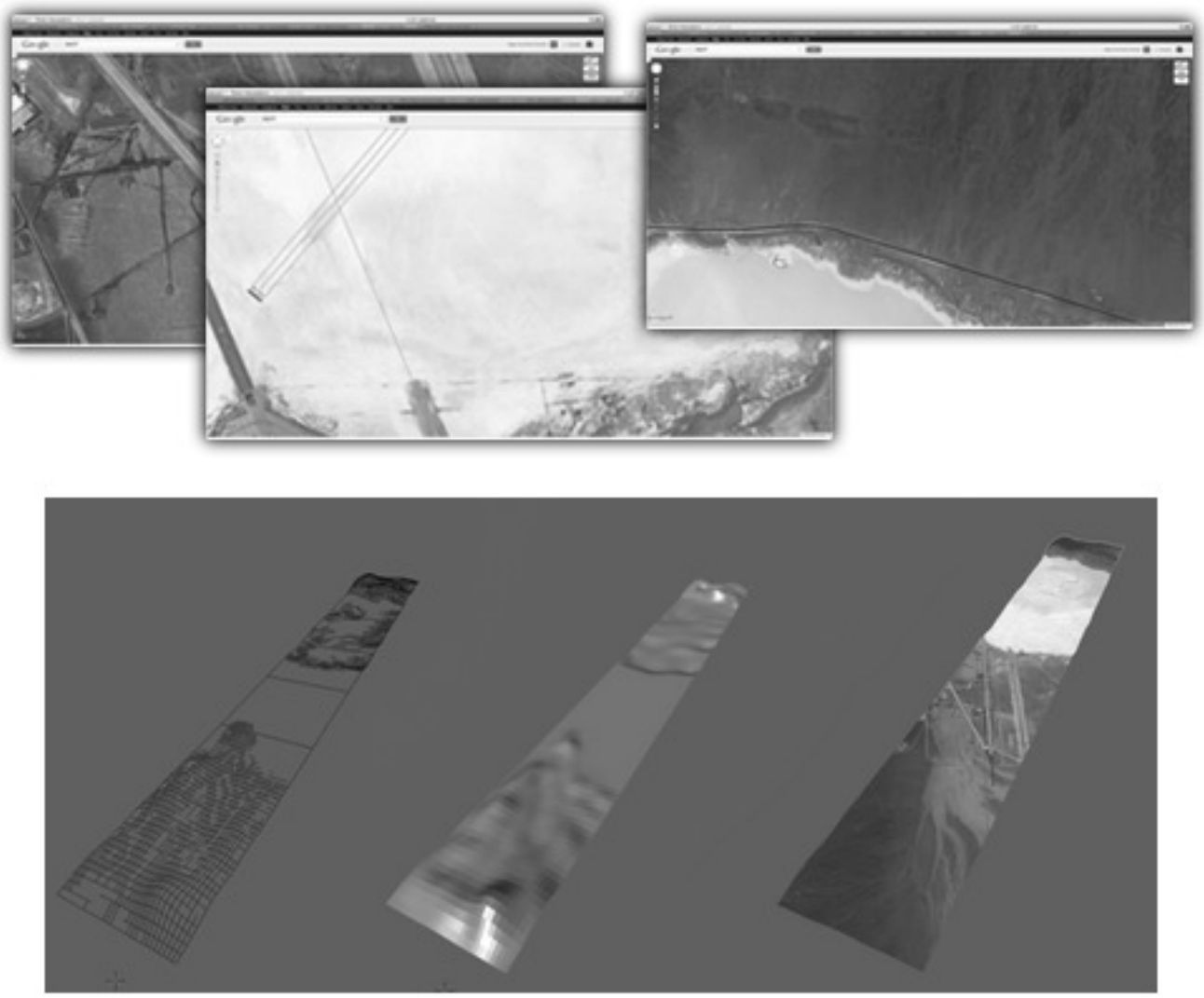


Figura 27.3: En nuestro ejemplo se han utilizado capturas de pantalla de google map para obtener texturas de terreno y se han mapeado sobre un plano en Blender. Posteriormente se ha utilizado el modo scuplt para dar relieve al terreno y generar un escenario tridimensional.

2. **Una única escena con elementos modificados dinámicamente**: Puede que en nuestro videojuego todos los niveles tengan el mismo tipo de elementos pero lo diferente sea la dificultad o el número de enemigos, en este caso podemos crear una única escena pero variar sus condiciones dinámicamente en función de en que nivel estemos.

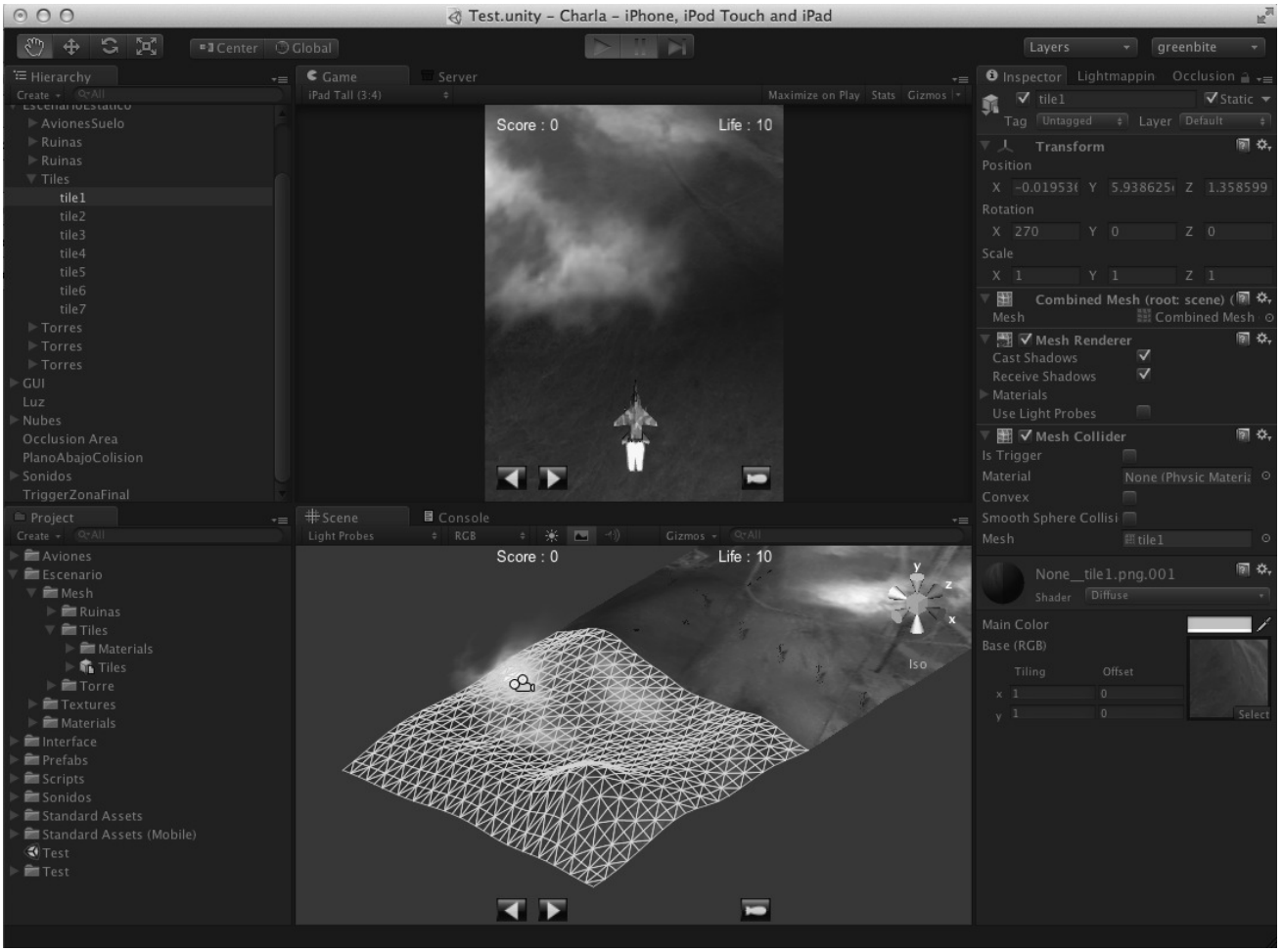


Figura 27.4: Interface de Unity3D. Dividida en las vistas más utilizadas: Jerarquía de escena, Assets del proyecto, Vista del videojuego, Vista de escena y Vista de propiedades del asset seleccionado.

Como las escenas pueden cargarse desde otra escena. Podemos realizar un cambio de escena cuando se ha llegado al final de un determinado nivel por ejemplo. Este cambio de escena mantendrá en memoria los elementos comunes como texturas o modelos 3D o sonidos que pertenezcan a los dos escenas, la escena que se descarga y la escena que se carga, por lo que dependiendo del caso esta carga suele ser bastante rápida. También podemos realizar los menús en una escena y desde ahí cargar la escena del primer nivel del juego.

Figura 27.5: Visualización de la jerarquía de assets de nuestro proyecto.

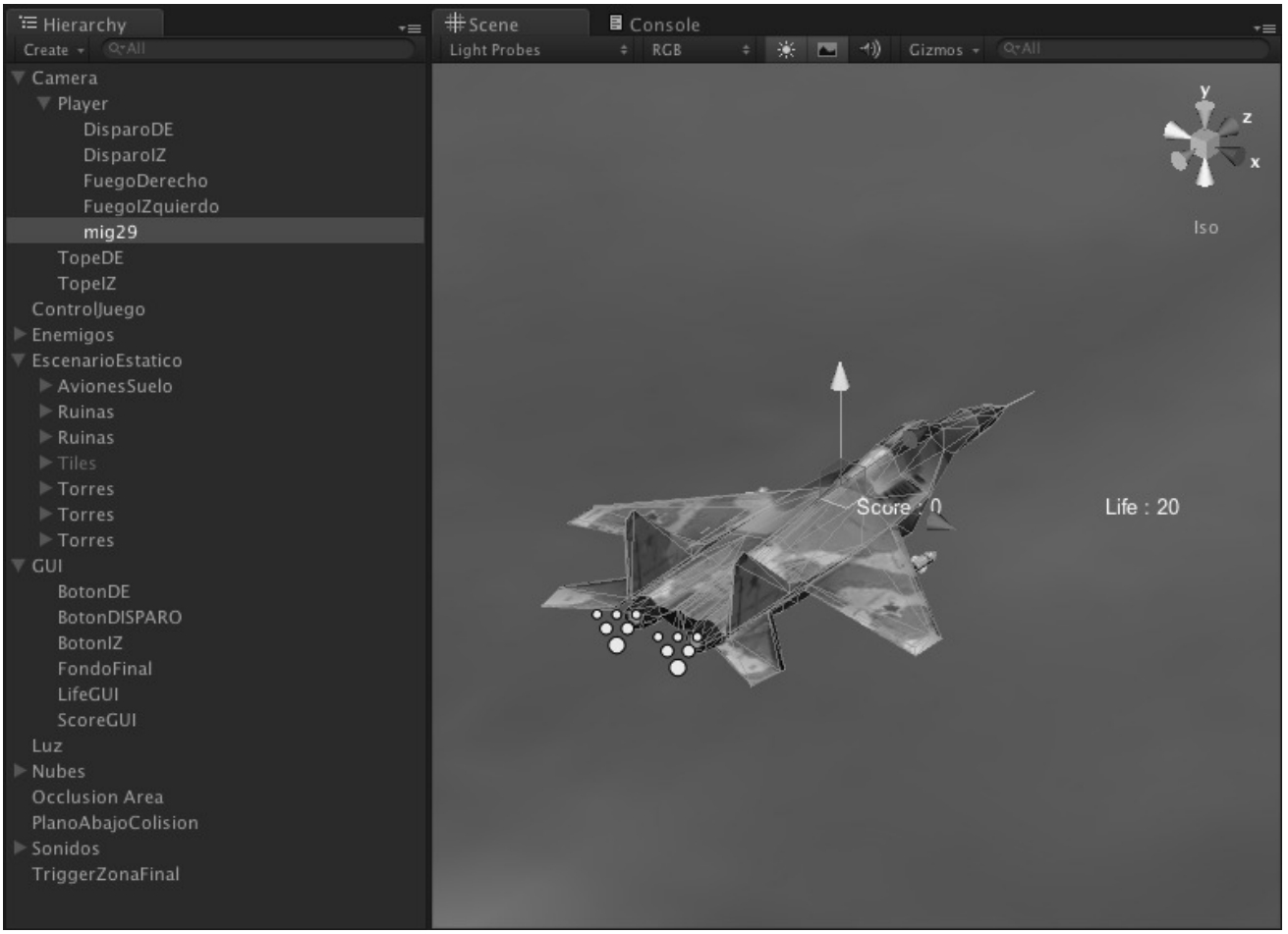


Figura 27.6: Para la escena de nuestro ejemplo hemos añadido el modelo 3D del escenario, los enemigos, el jugador y hemos establecido las relaciones de jerarquía necesarias entre estos elementos.

27.3. Creación de *prefabs*

Como se ha descrito en el apartado anterior, cada escena contiene instancias de objetos de nuestros Assets. Cada uno de los objetos de nuestra escena es un nodo, y cada nodo puede contener jerárquicamente a otros nodos.

Uso de *prefabs*

Cuando un elemento se repita en los diferentes niveles o en la misma escena debe de ser un prefab. De esta forma sólo se tiene una referencia de ese objeto y es óptimo en rendimiento y organización de assets.

Podemos agrupar esa jerarquía y darle un nombre propio para después serializarla e instanciarla en el futuro. A ese concepto se le conoce con el nombre de prefab. Podemos crear tantos prefabs como queramos a partir de jerarquías de objetos de una escena y son una parte fundamental para entender el método de trabajo con un motor.

En nuestra escena hemos creado prefabs para cada uno de los tipos de enemigos, y también hemos creado prefabs para el disparo, una explosión y un efecto de partículas de llamas.

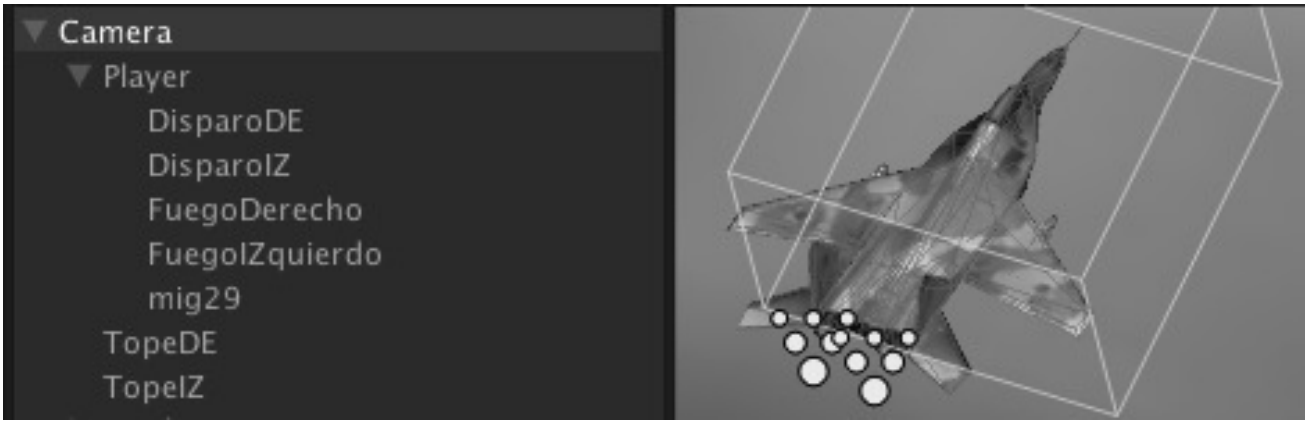


Figura 27.7: El nodo Camera contiene otros nodos de manera jerárquica.

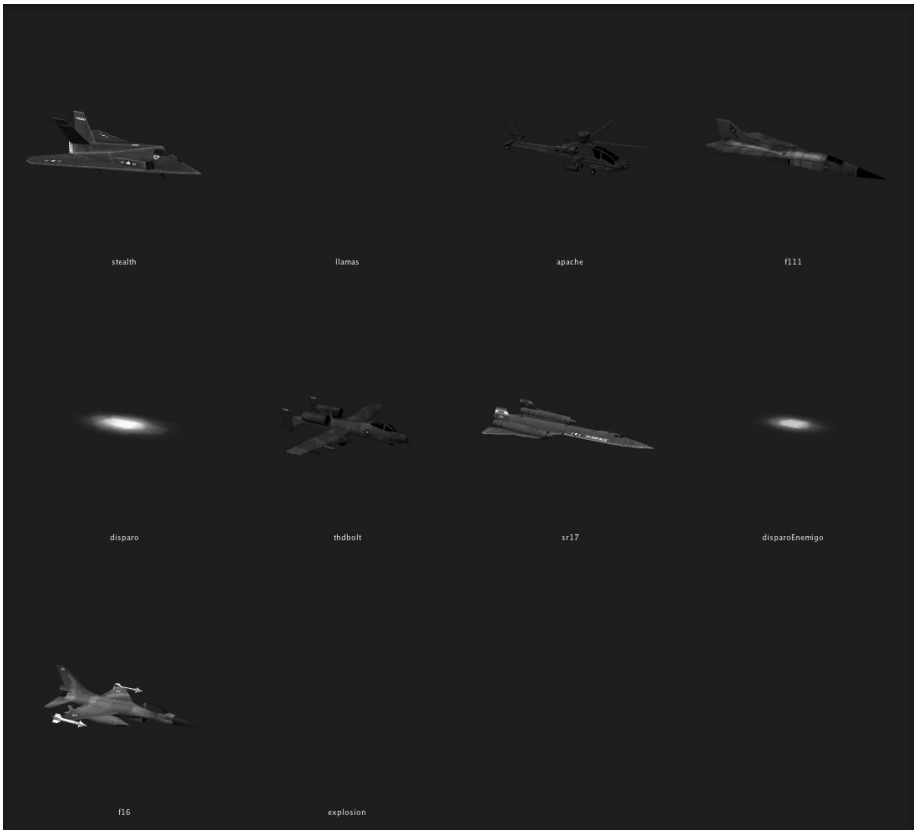


Figura 27.8: Prefabs de nuestro videojuego de ejemplo.

27.4. Programación de scripts

Una script es un fichero de código que contiene instrucciones sobre el comportamiento de un determinado actor de nuestra escena. Podemos añadir uno o varios scipts a cada uno de los elementos de nuestra escena y además los scripts tienen la posibilidad de hacer referencia a estos objetos o scripts de otros objetos.

Documentación

Podemos consultar la documentación de cada una de las API's para los tres lenguajes de scripting desde la página oficial de Unity3D

En los scripts podemos utilizar las clases y API's que nos proporciona el motor de videojuegos. Algunas de estas clases son:

- **GameObject**: Esta clase tiene información sobre el objeto. Todos los nodos de una escena son GameObjects.
- **Transform**: Esta clase representa la posición, rotación y escala de un elemento en el espacio tridimensional.
- **AudioSource**: Esta clase almacena un sonido y permite gestionar su reproducción.
- **Texture 2D**: Esta clase contiene una textura bidimensional.

Los scripts tienen algunos métodos especiales que podemos implementar como:

- **Update**: Este método es invocado por el motor gráfico cada vez que el objeto va a ser renderizado.
- **Start**: Este método es invocado por el motor gráfico cuando se instancia el objeto que contiene a este script.

Figura 27.9: En nuestro modelo del helicóptero se ha separado la hélice del resto del modelo para poder añadirle este script de movimiento.

27.4.1. Algunos scripts básicos

Algunos de nuestros enemigos son helicópteros. Podemos añadir el siguiente script a el objeto de las hélices para que realice una rotación sobre su eje perpendicular.

Listado 27.1: HeliceHelicoptero.js

```
1 #pragma strict
2
3 public var delta = 4.0;
4
5 function Update () {
6     //Rotar la helice en el eje y
7     transform.Rotate(0,Time.deltaTime * delta,0);
8 }
```

En nuestro escenario se han añadido nubes modeladas mediante planos y con una textura de una nube con transparencias. Para darle mayor realismo a este elemento se va a programar un script para mover las coordenadas u,v del mapeado de este plano provocando que la textura se mueve sobre el plano y simulando un movimiento de nubes.

Listado 27.2: Nube.js

```
1 #pragma strict
2
3 public var delta = 0.1;
4 public var moveFromLeftToRight : boolean = false;
```

```
5  private var offset : float = 0.0;
6
7  function Update () {
8      //Mover la coordenada u o v de la textura de el material del
           objeto que contiene este script
9      if (!moveFromLeftToRight) {
10         renderer.material.SetTextureOffset ("_MainTex", Vector2(
               offset,0));
11     } else {
12         renderer.material.SetTextureOffset ("_MainTex", Vector2(0,
               offset));
13     }
14     offset+=Time.deltaTime * delta;
15     if (offset>1.0) { offset-=1.0; }
16 }
```

En los scripts se utiliza el valor de Time.deltaTime para interpolar el valor de otros elementos con respecto al tiempo que ha pasado desde el último frame renderizado. De esta forma el videojuego va igual de rápido en todas las máquinas, pero se visualizará de manera más fluida en máquinas más rápidas debido a que al renderizar mayor numero de frames por segundo se producirán más posiciones intermedias de cada uno de los valores que dependen de Time.deltaTime.

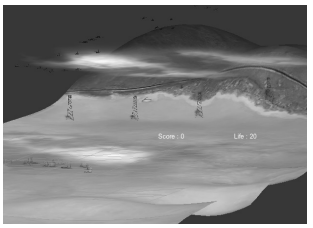


Figura 27.10: Visualización de las nubes.

27.4.2. *Triggers*

Un trigger es una porción del espacio definida por un objeto geométrico como una caja o una esfera que utilizaremos para colocar sobre la escena y de esta forma saber cuando un determinado objeto entra o sale de una zona concreta. De esta forma podremos invocar diferentes comportamientos en ese momento. En nuestra escena utilizaremos un trigger para determinar cuando hemos llegado al enemigo final.

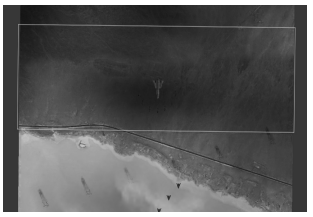


Figura 27.11: *Trigger* de la zona final.

Listado 27.3: TriggerFinal.js

```
1  #pragma strict
2
3  function OnTriggerEnter (other : Collider) {
4      if (other.tag=="Player"){
5          //Si entra un objeto dentro de el trigger y el objeto es el
               jugador, pasarle el mensaje a ControlJuego de que hemos
               llegado a la parte final
6          var ControlJuegoPointer : Transform = (GameObject.
               FindWithTag("ControlJuego")).transform;
7          var ControlJuegoStatic : ControlJuego = (
               ControlJuegoPointer.GetComponent("ControlJuego") as
               ControlJuego);
8          ControlJuegoStatic.EntrarEnZonaFinal();
9      }
10 }
```

27.4.3. Invocación de métodos retardada

A veces necesitaremos programar un evento para que pase trascurrido un determinado tiempo. En nuestro ejemplo esto ocurre con la explosión, que invocaremos su destrucción 5 segundos después de haberse instanciado.

Variables públicas

Las variables públicas nos permitirán modificar esos parámetros desde el interface de Unity3D cuando tengamos el objeto que contiene el script seleccionado.

Listado 27.4: DestruirPasadoUnTiempo.js

```
1  public var timeOut = 1.0;
2
```

```
3  function Start(){
4      //Realizar una llamada al método destruir pasados los segundos
           de timeOUT
5      Invoke ("Destruir", timeOut);
6  }
7
8  function Destruir(){
9      //Destruir el objeto que contiene este script
10     DestroyObject (gameObject);
11 }
```

27.4.4. Comunicación entre diferentes scripts

La mayoría de los scripts se comunicarán con otros scripts más complejos. Para ello hay que crear un puntero al objeto que contiene el script y utilizarlo para a su vez crear un puntero a la instancia de el script de ese objeto. Después, sobre el puntero de esa instancia de script, podremos invocar los métodos definidos en el mismo.

En el siguiente script que controla el disparo se utiliza este procedimiento de comunicación para invocar un método de el script Sonidos, encargado de la reproducción de los sonidos y músicas.

Sonido 3D

Es posible reproducir sonido en una posición del espacio para que el motor de juego calcule la atenuación, reberberación o efecto doppler del mismo.

Listado 27.5: Disparo.js

```
1  public var delta = 8.0;
2  public var timeOut = 5.0;
3  public var enemigo : boolean;
4
5  function Start() {
6      //Invocar PlaySonidoDisparo del script Sonidos del objeto
           Sonidos
7      var SonidosPointer : Transform = (GameObject.FindWithTag("
           Sonidos")).transform;
8      var SonidosStatic : Sonidos = (SonidosPointer.GetComponent("
           Sonidos") as Sonidos);
9      SonidosStatic.PlaySonidoDisparo();
10     Invoke ("Destruir", timeOut);
11 }
12
13 function Update () {
14     //Actualizar la posición del disparo
15     if (enemigo){
16         transform.position.z-=Time.deltaTime * delta*0.85;
17     } else {
18         transform.position.z+=Time.deltaTime * delta;
19     }
20 }
21 function OnCollisionEnter(collision : Collision) {
22     Destruir();
23 }
24 function Destruir() {
25     DestroyObject (gameObject);
26 }
```

Listado 27.6: Sonidos.js

```
1  #pragma strict
2
3  var SonidoDisparo : AudioSource;
4  var SonidoExplosionAire : AudioSource;
5  var SonidoExplosionSuelo : AudioSource;
6  var SonidoVuelo : AudioSource;
7  var MusicaJuego : AudioSource;
```

```
 8 var MusicaFinal : AudioSource;
 9
10 //Realizamos una fachada para que los demás objetos invoquen la
       reproducción de sonidos o música
11
12 function PlaySonidoExplosionSuelo(){
13     SonidoExplosionSuelo.Play();
14 }
15
16 function PlaySonidoExplosionAire(){
17     SonidoExplosionAire.Play();
18 }
19
20 function PlaySonidoDisparo(){
21     SonidoDisparo.Play();
22 }
23
24 function PlayMusicaJuego(){
25     MusicaJuego.Play();
26 }
27
28 function StopMusicaJuego(){
29     MusicaJuego.Stop();
30 }
31
32 function PlayMusicaFinal(){
33     MusicaFinal.Play();
34 }
35
36 function StopMusicaFinal(){
37     MusicaFinal.Stop();
38 }
39
40 function PlaySonidoVuelo(){
41     SonidoVuelo.Play();
42 }
```

27.4.5. Control del flujo general de la partida

Normalmente se suele utilizar un script que controla el flujo general de la partida. Este script se utiliza como nexo de unión entre el resto de los scripts y se le pasarán mensajes por ejemplo cuando ha terminado la partida. Podemos modelar el comportamiento de este script como si de un autómata se tratara.

Listado 27.7: ControlJuego.js

```
 1 #pragma strict
 2
 3 public var velocidadCamara :float = 2.0;
 4 public var enZonaFinal : boolean = false;
 5 public var Camara : Transform;
 6 public var ScoreGUI : GUIText;
 7 public var LifeGUI : GUIText;
 8
 9 public var BotonIZ : GUITexture;
10 public var BotonDE : GUITexture;
11 public var BotonDISPARO : GUITexture;
12
13 public var FondoFinal : GUITexture;
14 public var TexturaFinalBien : Texture2D;
15 public var TexturaFinalMal : Texture2D;
16
17 private var SonidosStatic : Sonidos;
18 private var Score : int;
19
```

Autómatas finitos

La gran mayoría de comportamientos de los actores de un videojuego pueden ser modelados como un autómata finito determinista.

Límite de FPS

En dispositivos móviles el limite de frames por segundo está ajustado por defecto a 30 FPS pero podemos cambiarlo modificando el atributo Application.targetFrameRate.

Retardos

Utilizaremos la instrucción WaitForSeconds para introducir retardos en los scripts.

```
20 function Awake(){
21     //Hacemos que el juego corra a 60 FPS como máximo
22     Application.targetFrameRate = 60;
23 }
24 
25 function Start(){
26     //Obtenemos el puntero a Sonidos y ajustamos algunos valores
           iniciales
27     var SonidosPointer : Transform = (GameObject.FindWithTag("
           Sonidos")).transform;
28     SonidosStatic = (SonidosPointer.GetComponent("Sonidos") as
           Sonidos);
29     SonidosStatic.PlayMusicaJuego();
30     SonidosStatic.PlaySonidoVuelo();
31     ScoreGUI.text="Score : "+Score;
32 }
33 
34 function Update () {
35     if (enZonaFinal && velocidadCamara>0.0){
36         //Si estamos en la zona final paramos el movimiento de
               manera gradual
37         velocidadCamara*=0.95;
38         if (velocidadCamara<0.1) { velocidadCamara=0; }
39     }
40 
41     if (velocidadCamara>0.0){
42         //Movemos la cámara en su componente z para hacer scroll
43         Camara.position.z+=Time.deltaTime * velocidadCamara;
44     }
45 }
46 
47 function EntrarEnZonaFinal(){
48     //Se ha entrado en el trigger de la zona final
49     enZonaFinal=true;
50 
51     SonidosStatic.StopMusicaJuego();
52     SonidosStatic.PlayMusicaFinal();
53 }
54 
55 function FinDeJuegoGanando(){
56     //Fin de partida cuando hemos completado la misión
57     FondoFinal.texture = TexturaFinalBien;
58     Restart();
59 }
60 
61 function FinDeJuegoPerdiendo(){
62     //Fin de partida cuando hemos fallado la misión
63     FondoFinal.texture = TexturaFinalMal;
64     Restart();
65 }
66 
67 function AddScore(valor : int){
68     //Añadimos puntos, por lo que hay que hacer la suma y
           actualizar el texto
69     Score+=valor;
70     ScoreGUI.text="Score : "+Score;
71 }
72 
73 function Restart(){
74     //Ocultamos los textos y botones
75     LifeGUI.enabled=false;
76     ScoreGUI.enabled=false;
77     BotonDISPARO.enabled=false;
78     BotonIZ.enabled=false;
79     BotonDE.enabled=false;
80     FondoFinal.enabled=true;
81 
82     //Esperamos 5 segundos y hacemos un reload de la escena
83     yield WaitForSeconds (5);
84     Application.LoadLevel(Application.loadedLevel);
85 }
```

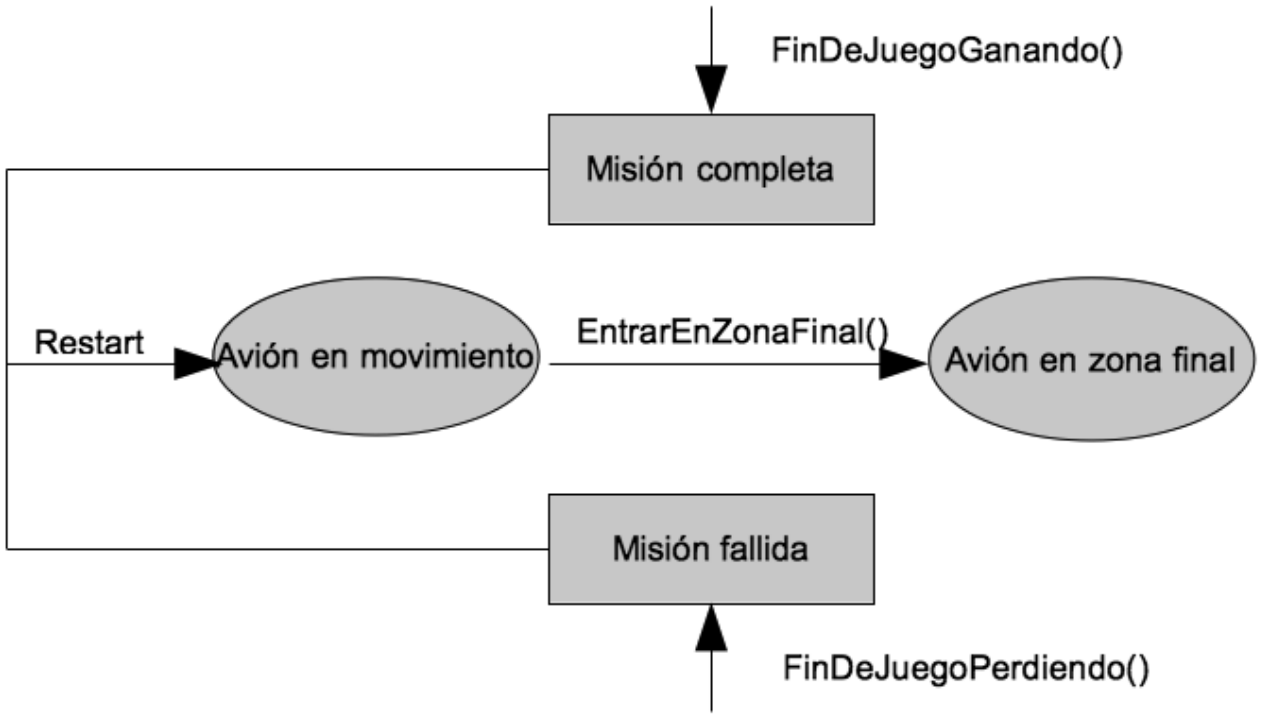


Figura 27.12: Diagrama de control de juego.

27.4.6. Programación de enemigos

Vamos a tener un único script para definir el comportamiento de todos los enemigos, incluidos el enemigo final. En este script vamos a definir una serie de atributos públicos que después ajustaremos con unos valores específicos para cada uno de los prefabs de enemigos.

Cada enemigo vendrá determinado un rango de disparo y un tiempo de recarga, y modificando estos parámetros crearemos enemigos más peligrosos que otros. Estos valores influirán en el calculo de puntuación que proporciona ese enemigo concreto al ser destruido.

Detección de colisiones

Cuando un objeto tridimensional tiene añadidos los elementos collider y rigidbody permite detectar colisiones mediante el método OnCollisionEnter.

Listado 27.8: Enemigo.js

```
#pragma strict

public var explosionPrefab : Transform;
public var llamasPrefab : Transform;
public var disparoPrefab : Transform;
public var player : Transform;
public var rangoDisparo : float;
public var tiempoRecarga = 0.5;
public var jefeFinal : boolean = false;

private var siguienteTiempoDisparo = 0.0;
private var enRangoDisparo : boolean=true;
```

```
13
14 private var llamasInstancia : Transform;
15 private var cantidadRotationCaida : Vector3 = Vector3(0,0,0);
16
17 function Update () {
18     if (transform.gameObject.rigidbody.useGravity){
19         //Si el enemigo está callendo, el avión rota sobre sus ejes
               porque entra en barrena
20         transform.Rotate(Time.deltaTime * cantidadRotationCaida.x,
               Time.deltaTime * cantidadRotationCaida.y,Time.deltaTime
                * cantidadRotationCaida.z);
21     } else {
22         if (player!=null){
23             var distancia : float = transform.position.z-player.
                   position.z;
24             if (distancia<=rangoDisparo && distancia>0) {
25                 //Si estamos en rango de disparo el avión dispara
                       al frente
26                 if (Time.time > siguienteTiempoDisparo){
27                     siguienteTiempoDisparo = Time.time +
                           tiempoRecarga;
28                     Instantiate(disparoPrefab, transform.position,
                           Quaternion.identity);
29                 }
30             }
31         }
32     }
33 }
34
35 function OnCollisionEnter(collision : Collision) {
36     //Determinar posición y rotación del punto de contacto de la
           colisión
37     var contact : ContactPoint = collision.contacts[0];
38     var rot : Quaternion = Quaternion.FromToRotation(Vector3.up,
           contact.normal);
39     var pos : Vector3 = contact.point;
40
41     var SonidosPointer : Transform = (GameObject.FindWithTag("
           Sonidos")).transform;
42     var SonidosStatic : Sonidos = (SonidosPointer.GetComponent("
           Sonidos") as Sonidos);
43
44     if (transform.gameObject.rigidbody.useGravity || collision.
           collider.tag=="Player"){
45         //Si estamos callendo y hemos vuelto a colisionar entonces
               explota
46         var ControlJuegoPointer : Transform = (GameObject.
               FindWithTag("ControlJuego")).transform;
47         var ControlJuegoStatic : ControlJuego = (
               ControlJuegoPointer.GetComponent("ControlJuego") as
               ControlJuego);
48
49         SonidosStatic.PlaySonidoExplosionSuelo();
50
51         //Instanciamos la explosión final en la posición del
               impacto
52         Instantiate(explosionPrefab, pos, rot);
53         if (llamasInstancia!=null){
54             Destroy (llamasInstancia.gameObject);
55         }
56
57         if (jefeFinal) {
58             ControlJuegoStatic.AddScore(500);
59             ControlJuegoStatic.FinDeJuegoGanando();
60         } else {
61             var cantidadScore : float =  (rangoDisparo * (1.0/
                   tiempoRecarga))*5;
62             ControlJuegoStatic.AddScore(cantidadScore);
63         }
64
```

```
65          //Eliminamos el objeto enemigo
66          Destroy (transform.gameObject);
67      } else if (collision.collider.tag=="Disparo"){
68          //Si no estamos callendo y hemos sido tocados por un
                disparo, empezamos a caer y a arder
69          SonidosStatic.PlaySonidoExplosionAire();
70
71          //Instanciamos llamas para la posición del impacto y las
                añadimos jerárquicamente al enemigo
72          llamasInstancia=Instantiate(llamasPrefab, transform.
                position, Quaternion.identity);
73          llamasInstancia.parent = transform;
74
75          //Activamos la gravedad del rigidBody del objeto
76          transform.gameObject.rigidbody.useGravity=true;
77
78          //Calculamos la cantidad de movimiento en caída para los
                ejes de manera aleatoria
79          cantidadRotationCaida.x=Random.Range(0, 20.0);
80          cantidadRotationCaida.y=Random.Range(0, 20.0);
81          cantidadRotationCaida.z=Random.Range(0, 20.0);
82      }
83  }
```

27.4.7. Programación del control del jugador

El jugador controlará su avión con los botones: izquierda, derecha y disparo. Además hay que tener en cuenta que cuando el jugador colisiona con un disparo enemigo o un enemigo debe reducir su vida, y cuando esta llega a cero explotar.

Listado 27.9: Player.js

```
1  #pragma strict
2
3  public var explosionPrefab : Transform;
4  public var disparoPrefab : Transform;
5  public var posicionDisparoIZ : Transform;
6  public var posicionDisparoDE : Transform;
7  public var tiempoRecarga = 0.5;
8  public var cantidadMovimiento = 0.1;
9  public var camara : Transform;
10 public var vida : int = 5;
11 public var LifeGUI : GUIText;
12 public var topeIZ : Transform;
13 public var topeDE : Transform;
14
15 private var siguienteTiempoDisparo = 0.0;
16 private var anteriorDisparoIZ : boolean = false;
17 private var botonIzquierda : boolean = false;
18 private var botonDerecha : boolean = false;
19 private var botonDisparo : boolean = false;
20
21 function Start(){
22     //Inicializamos el marcador de vida con el valor de vida
          inicial
23     LifeGUI.text="Life : "+vida;
24 }
25
26 function Update() {
27     if ((botonDisparo || Input.GetButton("Fire1")) && Time.time >
          siguienteTiempoDisparo){
28         //Si hay que disparar, instanciamos prefabs de disparo en
              las posiciones alternativamente izquierda y derecha de
              el avión del jugador
29         siguienteTiempoDisparo = Time.time + tiempoRecarga;
30         if (anteriorDisparoIZ){
```

Teclas de control

Aunque el juego final se controle mediante botones virtuales dibujados sobre la pantalla táctil del dispositivo, también permitiremos su control con un teclado para cuando probemos el juego en el emulador integrado en Unity3D.

```
31              Instantiate(disparoPrefab, posicionDisparoDE.position,
                    posicionDisparoDE.rotation);
32          } else {
33              Instantiate(disparoPrefab, posicionDisparoIZ.position,
                    posicionDisparoIZ.rotation);
34          }
35          anteriorDisparoIZ=!anteriorDisparoIZ;
36      }
37
38      if (botonIzquierda || Input.GetButton("Left")){
39          //Si hay moverse a la izquierda se actualiza la posición
                del jugador
40          //También se mueve un poco la cámara para simular un poco
                de efecto parallax
41          if (transform.position.x>topeIZ.position.x) {
42              transform.position.x-= Time.deltaTime *
                    cantidadMovimiento;
43              camara.position.x-= Time.deltaTime * cantidadMovimiento
                    /2;
44          }
45      } else if (botonDerecha || Input.GetButton("Right")) {
46          //Si hay moverse a la derecha se actualiza la posición del
                jugador
47          //También se mueve un poco la cámara para simular un poco
                de efecto parallax
48          if (transform.position.x<topeDE.position.x) {
49              transform.position.x+= Time.deltaTime *
                    cantidadMovimiento;
50              camara.position.x+= Time.deltaTime * cantidadMovimiento
                    /2;
51          }
52      }
53  }
54
55  function OnCollisionEnter(collision : Collision) {
56      if (collision.collider.tag=="DisparoEnemigo" || collision.
            collider.tag=="Enemigo"){
57          //Si el jugador colisiona con un disparo o un enemigo la
                vida disminuye
58          vida--;
59          LifeGUI.text="Life : "+vida;
60
61          if (vida<=0){
62              //Si la vida es 0 entonces acaba la partida
63              var ControlJuegoPointer : Transform = (GameObject.
                    FindWithTag("ControlJuego")).transform;
64              var ControlJuegoStatic : ControlJuego = (
                    ControlJuegoPointer.GetComponent("ControlJuego") as
                     ControlJuego);
65              ControlJuegoStatic.FinDeJuegoPerdiendo();
66
67              //Reproducimos sonido de explosión
68              var SonidosPointer : Transform = (GameObject.
                    FindWithTag("Sonidos")).transform;
69              var SonidosStatic : Sonidos = (SonidosPointer.
                    GetComponent("Sonidos") as Sonidos);
70              SonidosStatic.PlaySonidoExplosionSuelo();
71
72              //Instanciamos un prefab de explosión en la posición
                    del avión del jugador
73              Instantiate(explosionPrefab, transform.position,
                    Quaternion.identity);
74              //Eliminamos el avión del jugador
75              Destroy(gameObject);
76          }
77      }
78  }
79
80
81  //Métodos para controlar la pulsación de los botones virtuales
```

```
82
83 function ActivarBotonIzquierda(){
84     botonIzquierda=true;
85 }
86
87 function ActivarBotonDerecha(){
88     botonDerecha=true;
89 }
90
91 function ActivarBotonDisparo(){
92     botonDisparo=true;
93 }
94
95 function DesactivarBotonIzquierda(){
96     botonIzquierda=false;
97 }
98
99 function DesactivarBotonDerecha(){
100     botonDerecha=false;
101 }
102
103 function DesactivarBotonDisparo(){
104     botonDisparo=false;
105 }
```

27.4.8. Programación del *interface*

Utilizaremos botones dibujados sobre la pantalla táctil para controlar el videojuego. Para colocar cada uno de los botones virtuales utilizaremos un script que en función del valor del atributo tipoGUI lo posicionará en una zona determinada de la pantalla.

OnGUI

El método OnGui será llamado cuando se redimensiona la ventana o se cambia de resolución de modo que se calcule la posición de los elementos con respecto a las dimensiones de la pantalla para que siempre estén bien colocados.

Listado 27.10: ColocarGUI.js

```
1  #pragma strict
2
3  enum TipoGUI { Life, Score, BotonLeft, BotonRight, BotonShoot };
4  public var tipoGUI : TipoGUI;
5
6  function OnGUI () {
7      // Hacemos que el ancho del botón ocupe un 10 por ciento
8      // Alto del botón mantiene la proporción respecto a la imagen
9      var anchoBoton : float = Screen.width*0.1;
10     var altoBoton : float = anchoBoton * 94.0/117.0;
11     var margen : int = 10;
12
13     //Dependiendo del tipo de guiTexture o guiText; colocamos
14     switch(tipoGUI){
15         case tipoGUI.Life:
16             guiText.pixelOffset = Vector2 (Screen.width/2 - 55,
                   Screen.height/2 - margen);
17             break;
18         case tipoGUI.Score:
19             guiText.pixelOffset = Vector2 (-Screen.width/2 + margen
                   , Screen.height/2 - margen);
20             break;
21         case tipoGUI.BotonLeft:
22             guiTexture.pixelInset = Rect (-Screen.width/2 + margen,
                    -Screen.height/2 + margen, anchoBoton, altoBoton);
23             break;
24         case tipoGUI.BotonRight:
25             guiTexture.pixelInset = Rect (-Screen.width/2 +
                   anchoBoton+ 2*margen, -Screen.height/2 +margen,
                   anchoBoton, altoBoton);
26             break;
```

```
27          case tipoGUI.BotonShoot:
28              guiTexture.pixelInset = Rect (Screen.width/2 -
                    anchoBoton - margen, - Screen.height/2 + margen,
                    anchoBoton, altoBoton);
29              break;
30      }
31  }
```

Para darle funcionalidad a estos botones utilizaremos un único script, que en función de el valor de el atributo tipoGUI se comportará de un modo u otro cuando se pulse.

Touch

El objeto touch representa un toque sobre la pantalla y contiene información de si se está tocando o soltado, además de la posición x e y donde se realizó el toque.

Listado 27.11: BotonGUI.js

```
1  #pragma strict
2
3  public var tipoGUI : TipoGUI;
4  public var Boton : GUITexture;
5  public var TextureON : Texture2D;
6  public var TextureOFF : Texture2D;
7
8  private var wasClicked : boolean = false;
9  private var PlayerStatic : Player;
10
11 function Update(){
12      //Recorre los toques de pantalla
13      for (var touch : Touch in Input.touches) {
14          if (Boton.HitTest (touch.position)){
15              //Si algún toque está dentro de la zona del botón
16              if (touch.phase == TouchPhase.Began) {
17                  //Activar el botón cuando comienza el toque
18                  wasClicked = true;
19                  Activate();
20              } else if (touch.phase == TouchPhase.Ended || touch.
                        phase == TouchPhase.Canceled) {
21                  //Desactivar el botón cuando comienza el toque
22                  wasClicked = false;
23                  Deactivate();
24              }
25          }
26      }
27 }
28
29 function Activate() {
30      //Ponemos la textura botón pulsado
31      Boton.texture= TextureON;
32      //Dependiendo del tipo de botón que pulsamos enviamos el
            mensaje correspondiente a Player
33      switch(tipoGUI){
34          case tipoGUI.BotonLeft:
35                  PlayerStatic.ActivarBotonIzquierda();
36                  break;
37          case tipoGUI.BotonRight:
38                  PlayerStatic.ActivarBotonDerecha();
39                  break;
40          case tipoGUI.BotonShoot:
41                  PlayerStatic.ActivarBotonDisparo();
42                  break;
43      }
44 }
45
46 function Deactivate() {
47      //Ponemos la textura botón sin pulsar
48      Boton.texture= TextureOFF;
49      //Dependiendo del tipo de botón que soltamos enviamos el
            mensaje correspondiente a Player
50      wasClicked = false;
51          switch(tipoGUI){
```

```
52          case tipoGUI.BotonLeft:
53                  PlayerStatic.DesactivarBotonIzquierda();
54                  break;
55          case tipoGUI.BotonRight:
56                  PlayerStatic.DesactivarBotonDerecha();
57                  break;
58          case tipoGUI.BotonShoot:
59                  PlayerStatic.DesactivarBotonDisparo();
60                  break;
61      }
62  }
63
64  function Start () {
65      //Obtenemos el puntero a Player y ajustamos algunos valores
            iniciales
66      var PlayerPointer : Transform = (GameObject.FindWithTag("Player
            ")).transform;
67      PlayerStatic = (PlayerPointer.GetComponent("Player") as Player)
            ;
68      wasClicked = false;
69      Boton.texture= TextureOFF;
70  }
```

27.5. Optimización

El motor gráfico nos proporciona dos herramientas imprescindibles para optimizar nuestros videojuegos: lightmapping y occlusion culling. Aplicando estas técnicas reduciremos mucho la carga de renderizado y nos permitirá que nuestros videojuegos puedan correr a buena velocidad en dispositivos de poca potencia gráfica como smartphones o tablets.

27.5.1. *Light mapping*

Esta técnica consiste en calcular previamente las sombras que reciben y provocan los objetos estáticos de las escenas para generar unos mapas de sombreado que se aplican mediante multitextura sobre la maya de los modelos 3D. Los modelos sobre los que se aplica esta técnica son renderizados como polígonos con textura sin ningún sombreado, lo que evita el calculo de iluminación de la maya, ahorrando mucho tiempo de computo.

27.5.2. *Occlusion culling*

Esta técnica consiste en calcular previamente desde todas las posibles posiciones que puede tomar la cámara que objetos son visibles y cuales son ocluidos por otros. Después se utiliza esta información en tiempo de renderizado para no representar los objetos que después no van a ser visibles.

El cálculo de todas las posiciones que puede tomar la cámara se hace discretizando el espacio mediante una matriz tridimensional de la cual podremos elegir el nivel de granularidad. De este modo se calcula que objetos estáticos deberán más tarde ser renderizados cuando la cámara se encuentre en esta región del espacio.

La combinación de esta técnica junto con otras como frustrum culling permitirán que podamos tener escenas con millones de polígonos, pero en cada frame de renderizado sólo se representará una pequeña fracción de estos polígonos.

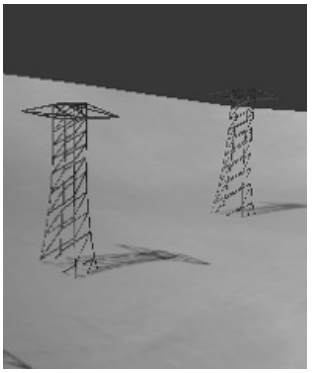


Figura 27.13: Visualización de la sombra provocada por una torre de electricidad.

Figura 27.15: Imagen de lo que visualiza el jugador cuando el motor gráfico está descartando objetos para no ser renderizados cuando la cámara pasa por la sección que se puede contemplar en la figura 27.18.

Figura 27.14: Ejemplo de las sombras que proyectan sobre el terreno los modelos tridimensionales de unas ruinas colocados en la escena.

Figura 27.16: Ejemplo de mapa generado mediante esta técnica que después se mapeará sobre el modelo 3D de la escena. Este mapa de sombreado es de la zona que se aprecia en la figura 27.18.

27.6. Resultado final

El resultado final es una pieza jugable de unos dos minutos y medio de duración que podría ser un nivel de un videojuego shoot em up de aviones con vista cenital.

Este videojuego está listo para compilarse para dispositivos con iOS o Android y funcionar en terminales de gama media-baja, pudiendo alcanzar los 60 FPS en terminales de gama alta.

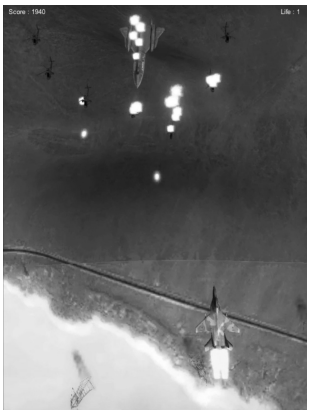


Figura 27.17: Imagen del enemigo final.

Figura 27.18: Imagen del resultado final.

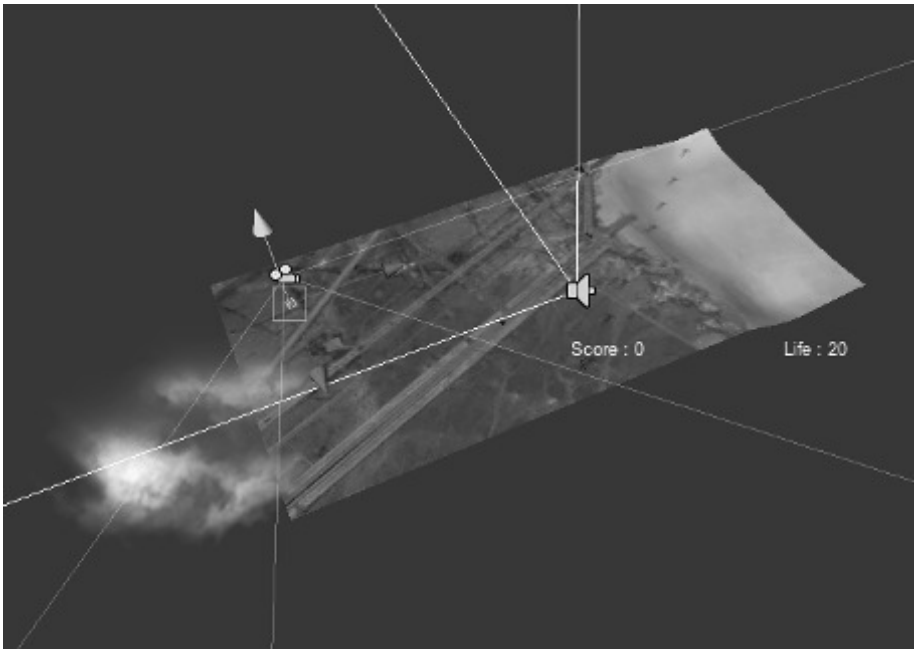


Figura 27.19: En nuestro ejemplo se ha dividido el escenario en porciones para poder aplicar la técnica, de este modo en un momento determinado sólo se renderiza una pequeña parte del escenario.

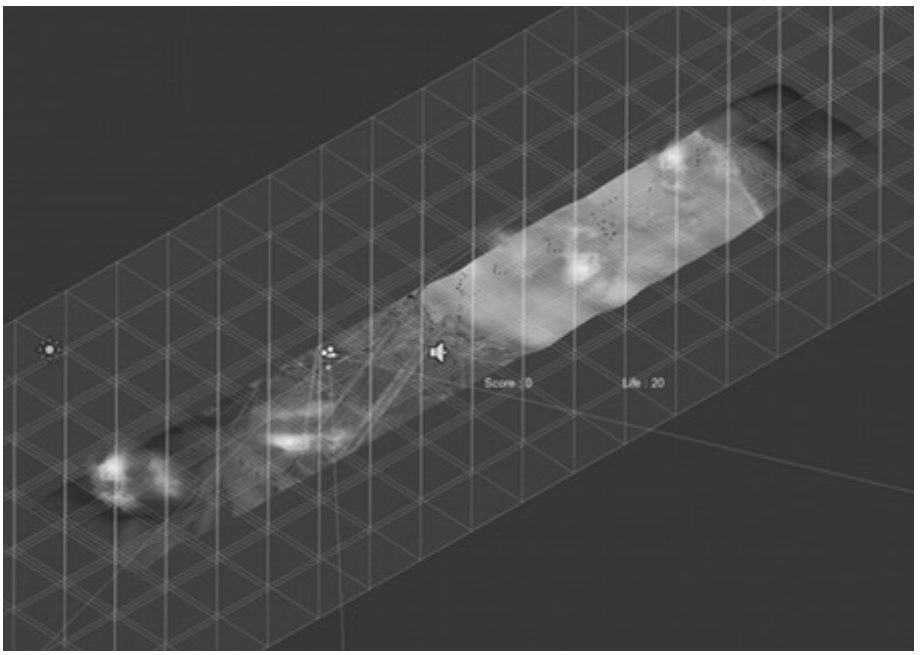
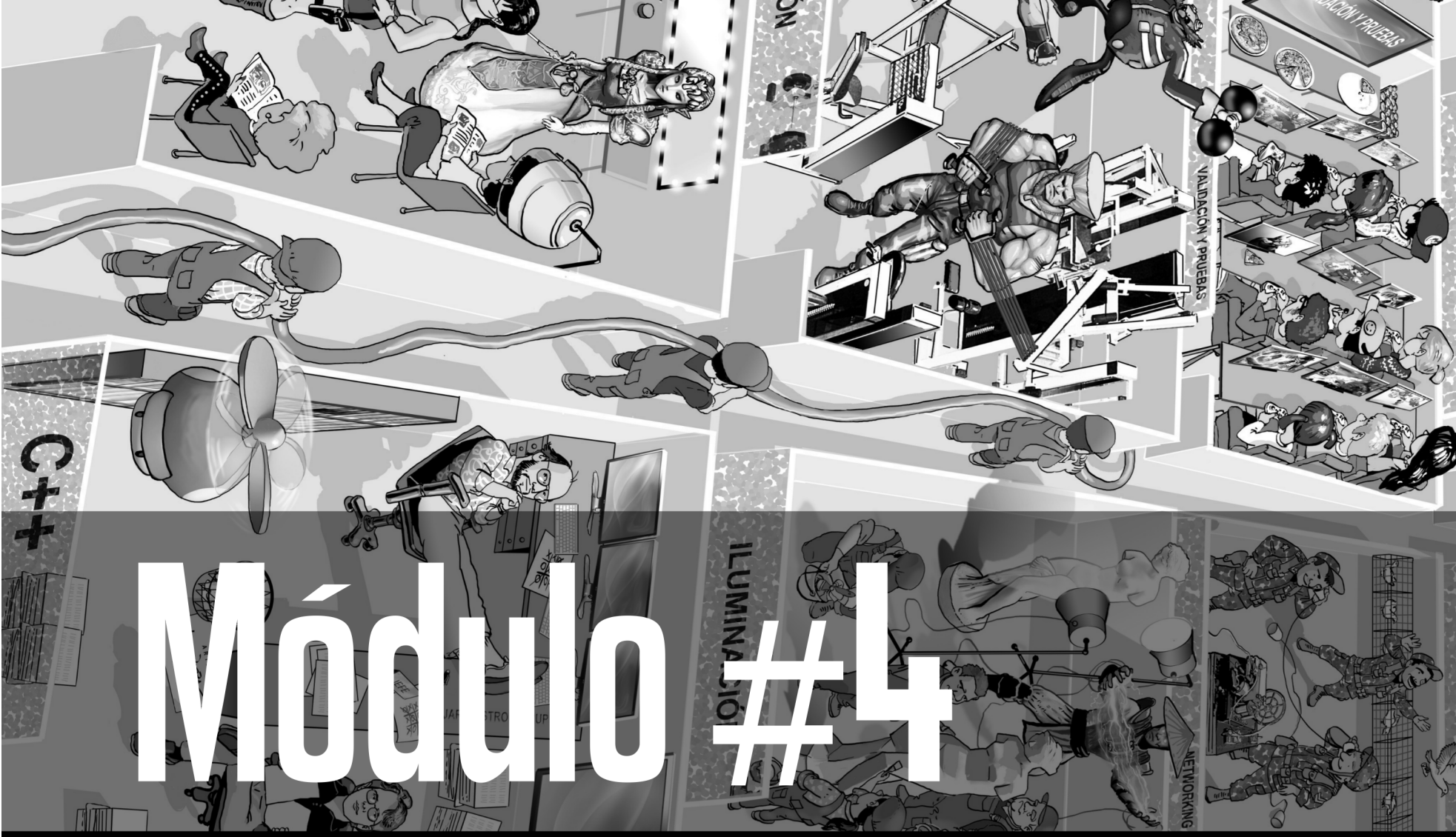


Figura 27.20: Nivel de granularidad elegido para la matriz de discretización del espacio en nuestro ejemplo.



Módulo #4

Desarrollo de Componentes

Desarrollo de Componentes

El objetivo de este módulo, titulado «*Desarrollo de Componentes*» dentro del Curso de Experto en Desarrollo de Videojuegos, consiste en profundizar en técnicas específicas vinculadas al desarrollo de videojuegos, como por ejemplo el uso de técnicas de Inteligencia Artificial o la programación multijugador en red. Para ello, una de las principales metas es la de complementar la visión general de la arquitectura de un motor de juegos con cuestiones específicas que resultan fundamentales para su desarrollo. Dentro del contexto de la Inteligencia Artificial, en este módulo se estudian técnicas fundamentales como la Lógica Difusa o los algoritmos genéricos, entre otras. Así mismo, se realiza una discusión del diseño orientado a agentes como pilar esencial en el desarrollo del componente inteligente de un videojuego. En la parte relativa al juego multijugador se exploran las posibilidades que ofrecen los sockets y, posteriormente, se discute cómo el uso de herramientas de más alto nivel, como los middlewares de comunicaciones pueden contribuir a facilitar el desarrollo del módulo de networking. Finalmente, este módulo también contempla aspectos relativos a la edición de audio, la gestión de vídeo y la importancia de la integración de nuevos dispositivos de interacción. En el contexto del desarrollo de videojuegos, técnicas como la visión por computador o la realidad aumentada pueden contribuir a mejorar la experiencia del jugador.

Capítulo 28

Inteligencia Artificial

Javier Alonso Albusac Jiménez
José Jesús Castro Sánchez
David Vallejo Fernández
Luis Jiménez Linares
Manuel Palomo Duarte

Figura 28.1: El robot-humanoide *Asimo*, creado por *Honda* en el año 2000, es uno de los exponentes más reconocidos de la aplicación de técnicas de IA sobre un prototipo físico real.

La Inteligencia Artificial (IA) es un elemento fundamental para dotar de realismo a un videojuego. Uno de los retos principales que se plantean a la hora de integrar comportamientos inteligentes es alcanzar un equilibrio entre la **sensación de inteligencia** y el tiempo de cómputo empleado por el subsistema de IA. Dicho equilibrio es esencial en el caso de los videojuegos, como exponente más representativo de las aplicaciones gráficas en tiempo real.

Este planteamiento gira, generalmente, en torno a la generación de soluciones que, sin ser óptimas, proporcionen una cierta sensación de inteligencia. En concreto, dichas soluciones deberían tener como meta que el jugador se enfrente a un reto que sea factible de manera que suponga un estímulo emocional y que consiga *engancharlo* al juego.

En los últimos años, la IA ha pasado de ser un componente secundario en el proceso de desarrollo de videojuegos a convertirse en uno de los aspectos más importantes. Actualmente, lograr un alto nivel de IA en un juego sigue siendo **uno de los retos** más emocionantes y complejos y, en ocasiones, sirve para diferenciar un juego normal de uno realmente deseado por los jugadores.

Tal es su importancia, que las grandes desarrolladores de videojuegos mantienen en su plantilla a ingenieros especializados en la parte de IA, donde los lenguajes de *scripting*, como Lua o Python, y la comunicación con el resto de programadores del juego resulta esencial.

En este capítulo se discuten cuestiones con gran relevancia en el ámbito de la IA aplicada a videojuegos, haciendo especial hincapié en las técnicas más consolidadas, como por ejemplo el uso de máquinas de estados finitas, el diseño basado en agentes, la aplicación de algoritmos de búsqueda o la lógica difusa. En la última sección del capítulo se discute un caso de estudio práctico enfocado a la IA.

28.1. Introducción a la IA para videojuegos

28.1.1. Aplicando el Test de Turing

La Inteligencia Artificial es un área fascinante y relativamente moderna de la Informática que gira en torno a la construcción de programas inteligentes. Existen diversas interpretaciones para el término *inteligente* (vea [79] para una discusión en profundidad), las cuales se diferencian en función de la similaritud con conceptos importantes como **racionalidad** y **razonamiento**.

De cualquier modo, una constante en el campo de la IA es la relación entre un programa de ordenador y el comportamiento del ser humano. Tradicionalmente, la IA se ha entendido como la intención de crear programas que actuasen como lo haría una persona ante una situación concreta en un contexto determinado.

Hace más de medio siglo, en 1950, Alan Turing propuso la denominada **Prueba de Turing**, basada en la incapacidad de una persona de distinguir entre hombre o máquina a la hora de evaluar un programa de ordenador. En concreto, un programa pasaría el test si un evaluador humano no fuera capaz de distinguir si las respuestas a una serie de preguntas formuladas eran o no de una persona. Hoy en día, esta prueba sigue siendo un reto muy exigente ya que, para superarlo, un programa tendría que ser capaz de procesar lenguaje natural, representar el conocimiento, razonar de manera automática y aprender.

Figura 28.2: Alan Turing (1912-1954), matemático, científico, criptógrafo y filósofo inglés, es considerado uno de los Padres de la Computación y uno de los precursores de la Informática Moderna.

Además de todas estas funcionalidades, esta prueba implica la necesidad de interactuar con el ser humano, por lo que es prácticamente imprescindible integrar técnicas de visión por computador y de robótica para superar la *Prueba Global de Turing*. Todas estas disciplinas cubren gran parte del campo de la IA, por lo que Turing merece un gran reconocimiento por plantear un problema que hoy en día sigue siendo un reto muy importante para la comunidad científica.

En el ámbito del **desarrollo de videojuegos**, la Prueba de Turing se podría utilizar para evaluar la IA de un juego. Básicamente, sería posible aplicar esta prueba a los *Non-Player Characters* (NPCs) con el objetivo de averiguar si el jugador humano es capaz de saber si son realmente *bots* o podrían confundirse con jugadores reales.

Aunque actualmente existen juegos que tienen un grado de IA muy sofisticado, en términos generales es relativamente fácil distinguir entre NPC y jugador real. Incluso en juegos tan trabajados desde el punto de vista computacional como el ajedrez, en ocasiones las decisiones tomadas por la máquina delatan su naturaleza.

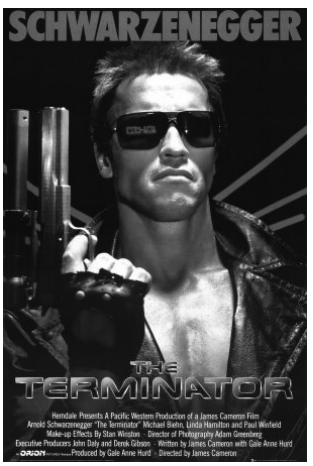


Figura 28.3: *Terminator* pasaría sin ninguna duda la *Prueba Global de Turing*, al menos hasta que tuviera que ir a la consulta del médico...

Desafortunadamente, los desarrolladores de videojuegos están condicionados por el tiempo, es decir, los videojuegos son aplicaciones gráficas en tiempo real que han de generar una determinada tasa de *frames* o imágenes por segundo. En otras palabras, este aspecto tan crítico hace que a veces el tiempo de cómputo dedicado al sistema de IA se vea reducido. La buena noticia es que, generalmente, el módulo responsable de la IA no se tiene que actualizar con la misma frecuencia, tan exigente, que el motor de *rendering*.

Aunque esta limitación se irá solventando con el incremento en las prestaciones hardware de las estaciones de juego, hoy en día es un gran condicionante que afecta a los recursos dedicados al módulo de IA. Una de las consecuencias de esta limitación es que dicho módulo se basa en proporcionar una ***ilusión de inteligencia***, es decir, está basado en un esquema que busca un equilibro entre garantizar la simplicidad computacional y proporcionar al jugador un verdadero reto.

28.1.2. Ilusión de inteligencia

De una manera casi inevitable, el *componente inteligente* de un juego está vinculado a la dificultad o al reto que al jugador se le plantea. Sin embargo, y debido a la naturaleza cognitiva de dicho componente, esta cuestión es totalmente subjetiva. Por lo tanto, gran parte de los desarrolladores opta por intentar que el jugador se sienta inmerso en lo que se podría denominar *ilusión de inteligencia*.

Por ejemplo, en el videojuego *Metal Gear Solid*, desarrollado por Konami y lanzado para PlayStation™en 1998, los enemigos empezaban a mirar de un lado a otro y a decir frases del tipo *¿Quién anda ahí?* si el personaje principal, Solid Snake, se dejaba ver mínimamente o hacía algún ruido en las inmediaciones de los NPCs. En este juego, el espionaje y la infiltración predominaban sobre la acción, por lo que este tipo de elementos generaban una cierta sensación de IA, aunque en realidad su implementación fuera sencilla.

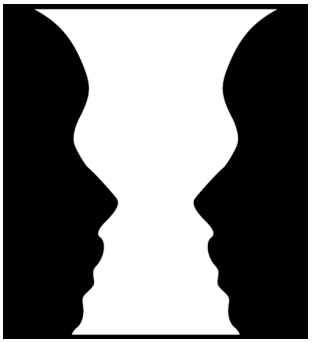


Figura 28.4: El jarrón de Rubin es una de las ilusiones ópticas más famosas. ¿Jarrón o dos caras?

Un caso más general está representado por modificar el **estado de los NPCs**, típicamente incrementando su nivel de *stamina* o vida. De este modo, el jugador puede tener la sensación de que el enemigo es más inteligente porque cuesta más abatirlo. Otra posible alternativa consiste en proporcionar más habilidades al enemigo, por ejemplo haciendo que se mueva más rápido o que dispare con mayor velocidad.

En [16], el autor discute el caso de la IA del juego *Halo*, desarrollado por *Bungie Studios* y publicado por *Microsoft Games Studio* en 2001, y cómo los desarrolladores consiguieron *engañar* a los *testers* del juego. En concreto, los desarrolladores asociaban el nivel de IA con la altitud de los puntos de impacto sobre los NPCs. Así, los jugadores percibían un grado bajo de IA cuando este nivel no era elevado, es decir, cuando los impactos en la parte inferior del NPC eran relevantes para acabar con el mismo. Sin embargo, al incrementar dicho nivel y forzar a los jugadores a apuntar a partes más elevadas del NPC, éstos percibían que el juego tenía una IA más elevada.

Desde un punto de vista general, el principal reto del componente de IA de un juego consiste en que el jugador se sienta inteligente, planteando una lucha equilibrada pero, al mismo tiempo, factible para el jugador.

Este planteamiento se puede implementar utilizando **polimorfismo**, es decir, haciendo uso de una implementación u otra en función del nivel de complejidad. La figura 28.5 muestra el diagrama de clases de una solución muy simplificada de este problema. La clase *Player* mantiene como estado la *stamina* y la altura del personaje.

Este nivel de *stamina* se verá reducido por los impactos de proyectiles sobre el personaje. Si la dificultad del juego es normal, todos los proyectiles tendrán el mismo impacto sobre la salud del personaje. Sin embargo, si la dificultad es elevada, entonces se hará una distinción explícita entre proyectiles que impactan en la parte superior del personaje y aquellos que lo hacen en la inferior.

El siguiente listado de código muestra una implementación muy simple de la clase *Player*.

Listado 28.1: Clase Player

```
#ifndef __PLAYER__
#define __PLAYER__

class Player {
 public:
  Player (int stamina, float height);
  ~Player () {}

  void reduceStamina (int value) {
    if ((_stamina - value) > 0)
      _stamina -= value;
    else
      destroy();
  }
  int getStamina () const { return _stamina; }
  int getHeight () const { return _height; }

 private:
  int _stamina;  // Vida del personaje.
  float _height; // Altura.
  void destroy () {}
};

#endif
```

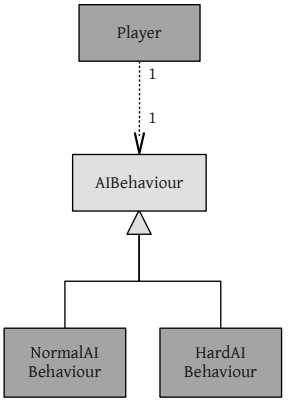


Figura 28.5: Diagrama de clases simplificado del ejemplo de *Halo*.

La parte más interesante de este ejemplo está en la clase abstracta *AIBehaviour*, la cual obliga a sus clases derivadas a implementar la función miembro *hit*, con el objetivo de evaluar su impacto en función del nivel de complejidad.

Listado 28.2: Clase AIBehaviour

```
#ifndef __AIBEHAVIOUR__
#define __AIBEHAVIOUR__

#include <OgreVector3.h>
#include "Player.h"

class AIBehaviour {
 public:
  AIBehaviour (Player* player) { _player = player; }

  // El impacto de hit dependerá del nivel de AI.
  virtual void hit (const Ogre::Vector3& bullet) = 0;
 protected:
  Player* _player;
};

class NormalAIBehaviour: public AIBehaviour {
 public:
  NormalAIBehaviour (Player* player): AIBehaviour(player) {}
  void hit (const Ogre::Vector3& bullet);
};

class HardAIBehaviour: public AIBehaviour {
 public:
  HardAIBehaviour (Player* player) : AIBehaviour(player) {}
  void hit (const Ogre::Vector3& bullet);
};

#endif
```

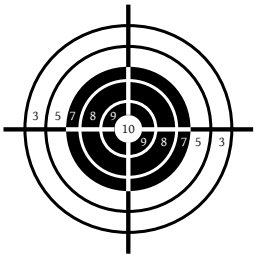


Figura 28.6: En los *shooters*, la zona de impacto sobre el enemigo es especialmente relevante y se puede usar para ajustar el nivel de dificultad.

La implementación de la función miembro *hit()* de las clases derivadas *NormalAIBehaviour* y *HardAIBehaviour* difiere en función del punto de impacto del proyectil sobre el personaje y, consecuentemente, en la reducción de su nivel de *stamina*.

Listado 28.3: Implementación de la función miembro hit

```
#include <AIBehaviour.h>

// SE OBVIA LA DETECCIÓN DE COLISIONES.
void
NormalAIBehaviour::hit
(const Ogre::Vector3& bullet)
{
  _player->reduceStamina(25.0);
}

// Considera la altura del impacto de la bala.
void
HardAIBehaviour::hit
(const Ogre::Vector3& bullet)
{
  float aux = _player->getHeight() / 3.0;
  float top = _player->getHeight() - aux;

  if (bullet.z > top)
    _player->reduceStamina(25.0);
  else
    _player->reduceStamina(10.0);
}
```

Como se puede apreciar, la reducción de vida sobre el personaje se mantiene constante para cualquier impacto de proyectil en el nivel de dificultad *normal*. Sin embargo, dicha reducción variará en el nivel de dificultad *hard*, dependiendo de la altura del impacto.

La idea de aplicar el Test de Turing a los NPCs de un juego plantea otro reto complejo y realmente atractivo para los desarrolladores de IA: la **cooperación**. Piense, por ejemplo, en una situación en la que el jugador se enfrenta a varios oponentes de manera simultánea en un *first-person shoooter*. El jugador que busca un reto ante dicha situación no espera que los NPCs ataquen de uno en uno por el mismo punto. Por el contrario, un jugador habilidoso esperaría un ataque coordinado, basado típicamente en una emboscada, tal y como un oponente real podría plantear en la mayoría de ocasiones.

IA cooperativa

La coordinación de comportamientos en los NPCs es una tarea compleja pero que, bien efectuada, puede incrementar notablemente la IA de un juego.

Evidentemente, una entidad siempre estará en desventaja ante un grupo, por lo que, tal y como se introdujo anteriormente, el desarrollador de IA ha de ser capaz de balancear la complejidad y saber adecuar una situación complicada en una situación que ofrezca un reto factible para el jugador, al menos en la mayoría de ocasiones.

28.1.3. ¿NPCs o Agentes?

En gran parte de la bibliografía del desarrollo de videojuegos, especialmente en la relativa a la IA, el concepto de *agent* (agente) se utiliza para referirse a las distintas entidades virtuales de un juego que, de alguna manera u otra, tienen asociadas un **comportamiento**. Dicho comportamiento puede ser trivial y basarse, por ejemplo, en un esquema totalmente preestablecido o realmente complejo y basarse en un esquema que gire entorno al aprendizaje. De cualquier modo, estas dos alternativas comparten la idea de mostrar algún tipo de inteligencia, aunque sea mínima.

Academic VS Game AI

Recuerde que en el ámbito académico, las técnicas de IA tienden a buscar soluciones óptimas, mientras que en el contexto de los videojuegos la tendencia general consiste en encontrar buenas soluciones.

Tradicionalmente, el **concepto de agente** en el ámbito de la IA se ha definido como cualquier entidad capaz de percibir lo que ocurre en el medio o contexto en el que habita, mediante la ayuda de sensores, y actuar en consencuancia, mediante la ayuda de actuadores, generando normalmente algún tipo de cambio en dicho medio o contexto. La figura 28.7 muestra esta idea tan general. Note cómo de nuevo la idea de la Prueba de Turing vuelve a acompañar al concepto de agente.

El concepto de agente se ha ligado a ciertas propiedades que se pueden trasladar perfectamente al ámbito del desarrollo de videojuegos y que se enumeran a continuación:

- **Autonomía**, de manera que un agente actúa sin la intervención directa de terceras partes. Por ejemplo, un personaje de un juego de rol tendrá sus propios deseos, de manera independiente al resto.
- **Habilidad social**, los agentes interactúan entre sí y se comunican para alcanzar un objetivo común. Por ejemplo, los NPCs de un *shooter* se comunicarán para cubrir el mayor número de entradas a un edificio.
- **Reactividad**, de manera que un agente actúa en función de las percepciones del entorno. Por ejemplo, un enemigo reaccionará, normalmente, atacando si es atacado.
- **Proactividad**, de manera que un agente puede tomar la iniciativa en lugar de ser puramente reactivo. Por ejemplo, un enemigo feroz atacará incluso cuando no haya sido previamente atacado.

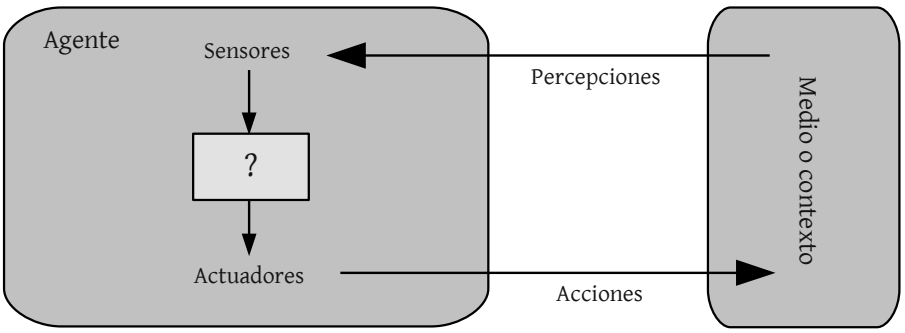


Figura 28.7: Visión abstracta delconcepto de agente

De manera adicional a estas propiedades, los conceptos de razonamiento y aprendizaje forman parte esencial del núcleo de un agente. Actualmente, existen juegos que basan parte de la IA de los NPCs en esquemas de aprendizaje y los usan para comportarse de manera similar a un jugador real. Este aprendizaje puede basar en la detección de patrones de comportamiento de dicho jugador.

Un ejemplo típico son los juegos deportivos. En este contexto, algunos juegos de fútbol pueden desarrollar patrones similares a los observados en el jugador real que los maneja. Por ejemplo, si la *máquina* detecta que el jugador real carga su juego por la parte central del terreno de juego, entonces podría contrarrestarlo atacando por las bandas, con el objetivo de desestabilizar al rival.

Aunque los agentes se pueden implementar haciendo uso de una filosofía basada en el diseño orientado a objetos, informalmente se suele afirmar que *los objetos lo hacen gratis, mientras que los agentes lo hacen porque quieren hacerlo*.

Comúnmente, los agentes basan su modelo de funcionamiento interno en una **máquina de estados**, tal y como se muestra de manera gráfica en la figura 28.8. Este esquema se ha utilizado durante muchos años como herramienta principal para proporcionar esa *ilusión de inteligencia* por parte del desarrollador de IA. De hecho, aunque en algunos proyectos se planteen arquitecturas mucho más sofisticadas, en la práctica la mayor parte de ellas girarán en torno a la idea que se discute en la siguiente sección.

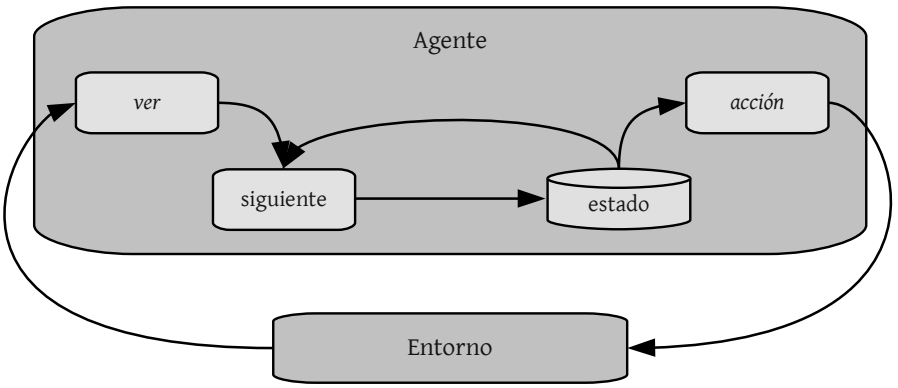


Figura 28.8: Visión abstracta del funcionamiento interno de un agente.

28.1.4. Diseño de agentes basado en estados

Acción, reacción

El modelo de diseño de las máquinas de estados está basado en el tradicional esquema de acción y reacción. Básicamente, ante el cumplimiento de una condición (acción) se produce una transición (reacción).

Una máquina de estados define el comportamiento que especifica las secuencias de estados por las que atraviesa un objeto durante su ciclo de ejecución en respuesta a una serie de eventos, junto con las respuestas a dichos eventos. En esencia, una máquina de estados permite descomponer el comportamiento general de un agente en *pedazos* o subestados más manejables. La figura 28.9 muestra un ejemplo concreto de máquinas de estados, utilizada para definir el comportamiento de un NPC en base a una serie de estados y las transiciones entre los mismos.

Como el lector ya habrá supuesto, los conceptos más importantes de una máquina de estados son dos: los estados y las transiciones. Por una parte, un **estado** define una condición o una situación durante la vida del agente, la cual satisface alguna condición o bien está vinculada a la realización de una acción o a la espera de un evento. Por otra parte, una **transición** define una relación entre dos estados, indicando lo que ha de ocurrir para pasar de un estado a otro. Los cambios de estado se producen cuando la transición se *dispara*, es decir, cuando se cumple la condición que permite pasar de un estado a otro.

Fuzzy Logic

La lógica difusa o borrosa es una técnica que permite tratar la incertidumbre y la vaguedad presenta en la mayoría de los problemas del mundo real. Ésta se puede utilizar para proporcionar más flexibilidad a un esquema basado en máquinas de estados.

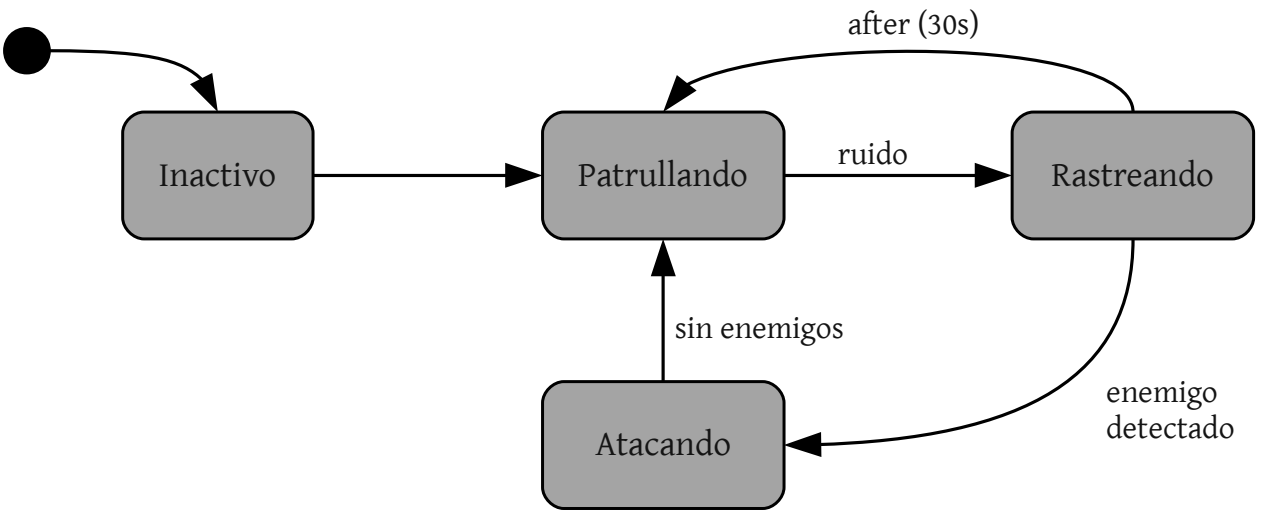


Figura 28.9: Máquina de estados que define el comportamiento de un NPC.

Aunque la idea general de las máquinas de estado es tremendamente sencilla, su popularidad en el área de los videojuegos es enorme debido a los siguientes factores [16]:

- Son **fáciles** de implementar y muy **rápidas**. Aunque existen diversas alternativas, todas ellas tienen una complejidad baja.
- Su **depuración** es sencilla, ya que se basan en el principio de descomposición en subestados que sean manejables.
- Tienen una **mínima sobrecarga computacional**, ya que giran en torno a un esquema *if-then-else*.
- Son muy **intuitivas**, ya que se asemejan al modelo de razonamiento del ser humano.
- Son muy **flexibles**, debido a que permiten la integración de nuevos estados sin tener un impacto significativo en el resto y posibilitan la combinación de otras técnicas clásicas de IA, como la lógica difusa o las redes neuronales.

Una máquina de estados se puede implementar utilizando distintas aproximaciones, como por ejemplo la que se estudió en el módulo 1, *Arquitectura del Motor*, para dar soporte al sistema de gestión de estados. En este contexto, es bastante común hacer uso del polimorfismo para manejar distintos estados que hereden de uno más general, proporcionando distintas implementaciones en función de dicha variedad de estados. En esencia, la idea consiste en mantener la interfaz pero concretando la implementación de cada estado. Normalmente, también se suele hacer uso del patrón *singleton* para manejar una única instancia de cada estado.

28.1.5. Los lenguajes de *script*

A medida que un proyecto de tanta envergadura como un motor de juegos o un juego comienza a crecer, el simple hecho de llevar a cabo una compilación para evaluar un cambio mínimo puede resultar desesperante debido a la cantidad de tiempo dedicada para generar un archivo ejecutable. La solución inmediata consiste en separar los datos de la implementación, es decir, plantear algún tipo de esquema que evite recompilar cuando los datos cambien.

Típicamente, esta solución se ha basado en hacer uso de **archivos de configuración** sobre los que leer, en tiempo de ejecución, valores globales o específicos de la configuración de algún módulo en particular. Este esquema tan sencillo es la esencia de los **lenguajes de *scripting***. En lugar de independizar solamente los datos, este tipo de lenguajes, en el ámbito del desarrollo de videojuegos, también permiten independizar gran parte de la lógica del juego. En este contexto, la lógica determina el comportamiento de las distintas entidades del juego a desarrollar, como por ejemplo los agentes o NPCs.

Figura 28.10: Actualmente, Python es uno de los lenguajes de *script* más populares debido a su gran potencia y a su facilidad de aprendizaje.

Este planteamiento representa la principal ventaja de los lenguajes de *scripting*, emplazándolos como una herramienta fundamental y muy potente en el desarrollo de proyectos software no triviales. Algunos ejemplos representativos de este tipo de lenguajes en el ámbito del desarrollo de videojuegos son Lua o Python.

Los lenguajes de *scripting* suelen ser **interpretados**. A diferencia de los lenguajes compilados, en los que se genera algún tipo de código máquina, los lenguajes interpretados se suelen leer, *parsear* e interpretar línea a línea en tiempo de ejecución, proceso que reduce el rendimiento de la aplicación. Ante esta problemática, algunos lenguajes de *scripting* son compilados.

Además de permitir independizar la lógica del juego, los lenguajes de *scripting* también se utilizan masivamente como herramientas de **prototipado rápido**. El principal objetivo que se persigue es el de obtener un producto o servicio que muestre la funcionalidad básica del sistema final, reduciendo drásticamente el tiempo y, por lo tanto, el coste del desarrollo. Posteriormente, se podría desarrollar dicho prototipo haciendo uso de un lenguaje compilado, como por ejemplo C++.

El uso de un lenguaje de *scripting* implica llevar a cabo un proceso de integración en el motor de juegos o en el juego en cuestión, normalmente desarrollados en C++. En el módulo M3, *Técnicas Avanzadas de Desarrollo*, se abordará esta problemática desde un punto de vista práctico.

El siguiente listado de código muestra una posible implementación del *juego de las chinas*[1] en Python. Como se puede apreciar, su sintaxis es muy legible y sencilla. Por ejemplo, sólo se necesita una línea de código (línea 11) para mostrar un mensaje por pantalla, que el usuario introduzca un dato y que dicho dato se almacene en una variable. El manejo de estructuras esenciales, como las listas, es tremendamente práctico (líneas 6 y 24), al igual que el tratamiento de excepciones (líneas 10 y 26). Python es un lenguaje de tipado dinámico, está soportado en un gran número de plataformas y es escalable gracias a una filosofía basada en módulos de desarrollo.

[1] `http://es.wikipedia.org/wiki/Chinos`

Listado 28.4: Implementación del juego de las chinas en Python

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sys, random

jugadas = []

def jugar ():
    while True:
        try:
            total_humano = int(input('\nTotal? '))

            n_maquina, total_maquina = predecir(total_humano)
            n_humano = int(input('Cuantas tenias? '))

            print '\nHumano: ' + str(n_humano)
            print 'Maquina: ' + str(n_maquina)
            if (n_maquina + n_humano) == total_humano:
                print 'Ganaste!'
            elif (n_maquina + n_humano) == total_maquina:
                print 'Perdiste!'

            # Para el módulo de IA...
            jugadas.append((n_humano, total_humano))

        except KeyboardInterrupt:
            print '\nSaliendo...'
            sys.exit(0)

def predecir (total_humano):
    n_maquina = random.randint(0, 3)
    total_maquina = random.randint(n_maquina, n_maquina + 3)
    print 'La maquina predice ' + str(total_maquina)
    return n_maquina, total_maquina

jugar()
```

El anterior listado de código contiene una función *predecir* que debería implementar una IA más sofisticada, más allá de devolver valores aleatorios. Una posible opción sería utilizar la estructura de datos *jugadas* para generar algún tipo de patrón que modelara el comportamiento del humano después de varias rondas de juego. Se plantea como ejercicio al lector la implementación de la función *predecir()* con el objetivo de estudiar la secuencia previa de predicciones por parte del jugador humano, con el objetivo de ganar un mayor número de partidas.

28.1.6. Caso de estudio. Un Tetris *inteligente*

En esta sección se discute cómo afrontar el módulo de IA del Tetris en el modo ***Human VS CPU***, es decir, cómo se puede diseñar el comportamiento de la máquina cuando se enfrenta a un jugador real.

Tradicionalmente, el modo *versus* del Tetris se juega a pantalla dividida, al igual que el modo de dos jugadores. En la parte izquierda juega el jugador real y en la derecha lo hace la máquina. En esencia, el perdedor es aquél que es incapaz de colocar una ficha debido a que ha ocupado la práctica totalidad del tablero sin *limpiar* líneas.

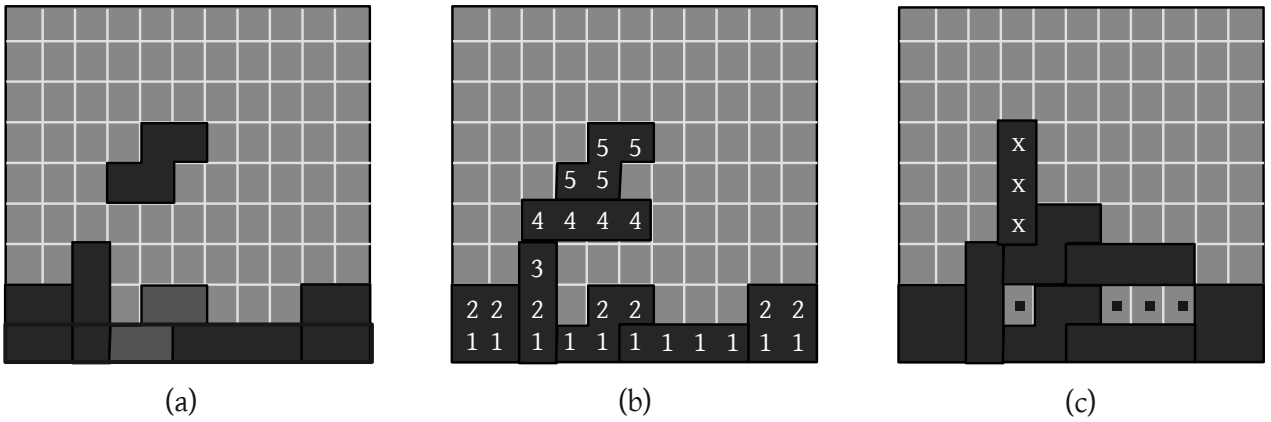


Figura 28.11: Planteando un módulo de IA para el Tetris. **a)** *Limpieza* de una línea, **b)** Cuantificando la altura del montón de fichas, **c)** Problema de huecos perdidos (los puntos representan los huecos mientras que las 'x' representan los potenciales bloqueos).

Para llevar a cabo la gestión de la dificultad de este modo de juego o, desde otro punto de vista, modelar la habilidad de la máquina, se pueden plantear diversas alternativas. Por ejemplo, si se desea modelar un nivel de complejidad elevado, entonces bastaría con incrementar la **velocidad** de caída de las fichas. En este caso, la máquina no se vería afectada ya que tendría tiempo más que suficiente para colocar la siguiente ficha. Sin embargo, el jugador humano sí que se vería afectado significativamente.

Otra posibilidad para ajustar el nivel de dificultad consistiría en que el jugador y la máquina recibieran distintos **tipos de fichas**, computando cuáles pueden ser más adecuadas para completar una línea y, así, reducir la altura del montón de fichas. También sería posible establecer un ***handicap***, basado en introducir deliberadamente piezas en la pantalla del jugador real.

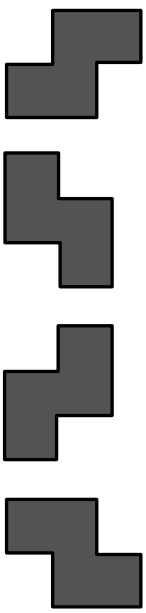


Figura 28.12: Cuatro posibles rotaciones de una ficha de Tetris (2D).

No obstante, la implementación de todas estas alternativas es trivial. El **verdadero reto** está en modelar la IA de la máquina, es decir, en diseñar e implementar el comportamiento de la máquina a la hora de ir colocando fichas.

La solución inicial planteada en esta sección consiste en **asignar una puntuación** a cada una de las posible colocaciones de una ficha. Note que las fichas se pueden rotar y, al mismo tiempo, se pueden colocar en distintas posiciones del tablero. El objetivo perseguido por el módulo de IA será el de colocar una ficha allí donde obtenga una mejor puntuación. El siguiente paso es, por lo tanto, pensar en cómo calcular dicha puntuación.

Para ello, una opción bastante directa consiste en distinguir qué aspectos resultan fundamentales para ganar o perder una partida. En principio, el módulo de IA debería evitar formar torres de fichas de gran altura, ya que lo aproximarían a perder la partida de forma inminente, tal y como muestra la figura 28.11.b. Este factor debería suponer una penalización para la puntuación asociada a colocar una ficha. Por el contrario, la máquina debería *limpiar* líneas siempre que fuera posible, con el objetivo de obtener puntos y evitar que el montón de fichas siga creciendo. Este factor representaría una bonificación.

Además, idealmente el módulo de IA debería evitar generar espacios que no se puedan aprovechar por las fichas siguientes, es decir, debería evitar que se perdieran huecos, tal y como se refleja en la figura 28.11.c. Evidentemente, la generación de huecos perdidos es, a veces, inevitable. Si se produce esta situación, el módulo de IA debería evitar la colocación de fichas sobre dichos huecos, con el objetivo de liberarlos cuanto antes y, en consecuencia, seguir *limpiando* líneas.

Estos cuatro factores se pueden ponderar para construir una **primera aproximación** de la fórmula que se podría utilizar para obtener la puntación asociada a la colocación de una ficha:

$$P = w_1 * salt + w_2 * nclears + w_3 * nh + w_4 * nb \tag{28.1}$$

donde:

- $salt$ representa la suma de las alturas asociada a la pieza a colocar en una determinada posición,
- $nclears$ representa el número de líneas *limpiadas*,
- nh representa el número de huecos generados por la colocación de la ficha en una determinada posición,
- nb representa el número de bloqueos como consecuencia de la potencial colocación de la ficha.
- w_i representa el peso asociado al factor i.

Evidentemente, $nclears$ tendrá asociado un peso positivo, mientras que el resto de factores tendrán asociados pesos negativos, ya que representan penalizaciones a la hora de colocar una ficha.

Una inconveniente inmediato de esta primera aproximación es que no se contempla, de manera explícita, la construcción de **bloques consistentes** por parte del módulo de IA de la máquina. Es decir, sería necesario incluir la lógica necesaria para premiar el acoplamiento entre fichas de manera que se premiara de alguna forma la *colocación lógica* de las mismas. La figura 28.13 muestra un ejemplo gráfico de esta problemática, cuya solución resulta fundamental para dotar a la máquina de un comportamiento *inteligente*.

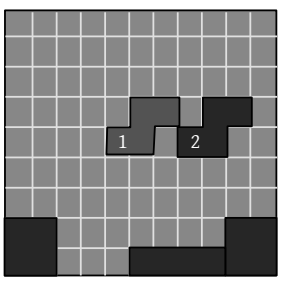


Figura 28.13: Idealmente, la posición 2 debería ser premiada en detrimento de la posición 1, ya que facilita la colocación de futuras piezas.

28.2. Técnicas Fundamentales

En esta sección se describirán brevemente algunas de las técnicas más extendidas en el ámbito de la Inteligencia Artificial que se emplean en el desarrollo de videojuegos, como son la lógica difusa, los algoritmos genéticos y las redes neuronales. La **lógica difusa** permite presentar al computador una forma alternativa de solucionar los problemas de una forma similar a como lo hacen los humanos, mediante el uso de términos y conceptos lingüísticos; así como el tratamiento de la imprecisión y vaguedad presentes en multitud de problemas reales.

Por otro lado, los **algoritmos genéticos** están basados en la teoría de la evolución y permiten encontrar de forma eficiente soluciones efectivas (no tiene por qué ser la óptima) en poblaciones de individuos amplias (o espacio de soluciones). Para ello, los algoritmos genéticos emplean varias técnicas de combinación, selección y mutación que permiten llegar a soluciones finales mediante la combinación de soluciones parciales.

Una de las principales características potenciales de este tipo de algoritmos es que son adaptativos; permiten encontrar soluciones en entornos dinámicos o cambiantes, incluso si existe un gran desconocimiento inicial sobre el problema a tratar. Esta característica los hace especialmente adecuados en el desarrollo de videojuegos de última generación en donde se busca el mayor realismo posible, y se intenta modelar entornos y comportamientos poco predecibles.

Por otro lado, las **redes neuronales** también están basadas en la propia naturaleza. Se trata de algoritmos de aprendizaje y razonamiento basados en el sistema nervioso de los animales (interconexión de neuronas en una red que colabora para producir un estímulo de salida). Las redes neuronales y los algoritmos genéticos se suelen combinar con frecuencia, ya que los segundos permiten construir el espacio de casos o entrenamiento que las redes neuronales necesitan para su correcto funcionamiento.

28.2.1. Lógica Difusa

La lógica difusa es un modelo matemático creado por el profesor Lotfi Zadeh de la Universidad de Berkeley en California (EEUU). El profesor Zadeh presentó la teoría de conjuntos difusos por primera vez en el año 1965 mediante un artículo publicado en la revista *Information and Control* [105]. Según su autor, la lógica difusa se basa en dos principios fundamentales [14]:

- Permite presentar al computador una forma alternativa de solucionar los problemas, de una forma similar a como lo hacen los humanos.
- La mayoría de elementos se representan en la lógica difusa mediante grados de pertenencia.

Para aquellos que no estén familiarizados con esta técnica, los dos principios anteriores pueden resultar un tanto confusos; desarrollemos con mayor detalle la principal idea en la que se basa cada uno de ellos.

Figura 28.14: Profesor Lofti Zadeh, padre de la lógica difusa, presentó su teoría en el año 1965 en la revista *Information and Control*.

En cuanto al primero, los computadores son máquinas que trabajan de forma metódica y que resuelven los problemas con total exactitud. Normalmente, un computador necesita unos parámetros de entrada, los procesa y genera unos datos de salida como solución a un problema. Sin embargo, los humanos suelen analizar las situaciones o resolver los problemas de una manera más imprecisa, sin la necesidad de conocer todos los parámetros o que la información relativa a los mismos esté completa.

Por ejemplo, supongamos el caso en el que dos personas se están pasando una pelota. El lanzador no es consciente en ningún momento, del ángulo que forma su brazo con el cuerpo, ni la fuerza exacta que realiza para lanzar la bola. Uno de ellos podría pedir al otro que le lanzara la pelota más fuerte. El término lingüístico *más fuerte* es impreciso en sí; en ningún momento se especifica cuanto debe aumentar el valor de la fuerza para cumplir el objetivo deseado. La persona encargada de lanzar, puede resolver el problema rápidamente, aumentando la fuerza con respecto al lanzamiento anterior sin la necesidad de realizar cálculos precisos. Esta forma de trabajar, en la que se manejan términos lingüísticos, conceptos vagos e imprecisos es un tanto opuesta a la forma en la que el computador trabaja. Sin embargo, puede ofrecer soluciones rápidas y eficientes a problemas reales, con un bajo coste computacional.

Por otro lado, el segundo principio enuncia que la lógica difusa se basa principalmente en el estudio de los grados de pertenencia a los conjuntos difusos definidos. Normalmente, un sistema difuso consta de un conjunto de variables de entrada; cada una de estas variables posee un dominio de definición (rango de valores que las va-

riables pueden tomar) en el que se incluyen los conjuntos difusos. Según el valor de entrada de la variable, éste pertenecerá a uno o varios conjuntos del dominio con un cierto grado de pertenencia. Esta es una de las principales características que diferencian a la lógica difusa de la lógica tradicional.

En la lógica bivaluada existen dos valores de verdad posibles: verdadero y falso. En la lógica difusa pueden existir valores intermedios, es decir, un hecho podría ser no del todo cierto y no del todo falso. En cuanto a la teoría de conjuntos tradicional, un elemento pertenece absolutamente a un conjunto o no pertenece. En el caso de la lógica difusa, un elemento puede pertenecer a diferentes conjuntos con distintos grados. Los grados de pertenencia pertenecen al intervalo [0,1], donde 1 representa pertenencia absoluta y 0 el caso contrario. Por tanto, podemos decir que la lógica difusa es una generalización de la lógica clásica.

Veamos un ejemplo; supongamos que un sistema posee una variable de entrada llamada altura. El objetivo de este sistema podría ser catalogar a un conjunto de personas adultas como muy bajo, bajo, mediano, alto o muy alto. Supongamos que la altura puede variar entre 1.40 y 2.20. En el dominio de definición de la variable altura habrá que indicar de algún modo la correspondencia entre los términos *muy bajo, bajo, mediano, alto y muy alto*, y el rango posible de valores numéricos [1.40-2.20]. Supongamos que en un sistema clásico, se considera personas bajas hasta 1.69 y las personas medianas desde 1.70 hasta 1.75. En este sistema, una persona que mida 1.69 sería clasificada como baja, mientras que una persona que mida 1.70 sería clasificada como mediana.

Tal como se puede apreciar, la discriminación en el proceso de clasificación entre una persona y otra es excesiva teniendo en cuenta que la diferencia es mínima (sólo un centímetro). La lógica difusa contempla la pertenencia a varios conjuntos al mismo tiempo, es decir, una persona que mida 1.70 podría pertenecer al conjunto de los medianos con un grado 0.6 mientras que al conjunto de los bajos podría ser 0.4. Este es un ejemplo orientativo para concebir una idea general sobre la esencia en la que se basa la lógica difusa; en secciones posteriores se explicará con mayor detalle la definición de conjuntos difusos y el cálculo de grados de pertenencia.

La lógica difusa permite resolver problemas reales de forma imprecisa con el uso de términos lingüísticos.

Aplicaciones generales de la lógica difusa

Desde el año 1965, la lógica difusa se ha implantado con éxito en multitud de ámbitos, teniendo una mayor acogida inicial en Japón en la fabricación de electrodomésticos y el control industrial. Algunos ejemplos de aplicación son los siguientes:

- Sistema de control inteligente de aires acondicionados.
- Enfoque automático en cámaras digitales.
- Electrodomésticos (frigoríficos, lavadoras, ...).
- Control industrial.
- Comercio electrónico.

- Sistemas de escritura.
- Mejora de la eficiencia en el consumo de combustibles en vehículos.
- Sistemas expertos que simulan el comportamiento humano.
- Tecnología informática.

Figura 28.15: La lógica difusa se ha empleado con frecuencia para controlar el movimiento de aeroplanos en simuladores de vuelo.

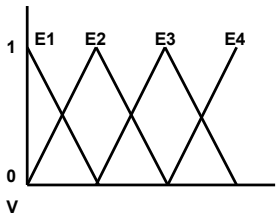


Figura 28.16: Definición de dominio difuso de la variable V mediante conjuntos triangulares.

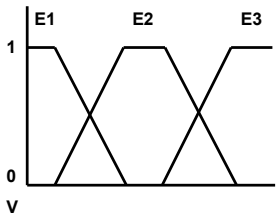


Figura 28.17: Definición de dominio difuso de la variable V mediante conjuntos trapezoidales.

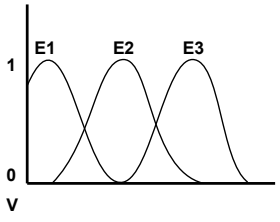


Figura 28.18: Definición de dominio difuso de la variable V mediante conjuntos curvilíneos.

De los ámbitos anteriores, el control inteligente de aparatos de aire acondicionado es uno de los más frecuentes en donde se aplica con éxito la lógica difusa. Estos aparatos disponen de un termostato que mide la temperatura ambiente y, en función de ésta, el aparato se activa con mayor o menor potencia para obtener la temperatura deseada. La aplicación de la lógica difusa en este caso evita activaciones múltiples cuando la temperatura oscila frecuentemente entre los límites establecidos en el termostato. De esta forma, se permite un mayor ahorro energético y, en consecuencia, un ahorro económico importante. Este tipo de sistemas no sólo permiten un control suavizado de la activación del suministro de aire, sino también del grado y potencia con la que se emite el aire para conseguir la temperatura deseada.

La lógica difusa también es realmente útil para la toma de decisiones [64]; un ejemplo de ello es el comercio electrónico. En numerosas ocasiones, un cliente no tiene claro las características del producto que desea adquirir; de esta forma, en el proceso de negociación se manejan términos imprecisos, subjetivos o vagos. La lógica difusa es un modelo que, por su propia naturaleza, se adapta correctamente a este tipo de situaciones y facilita la toma de decisiones.

En general, el uso de la lógica difusa es conveniente cuando se trata un proceso complejo no lineal y en el que se maneja conocimiento que no está estrictamente definido.

La lógica difusa en el desarrollo de videojuegos

La lógica difusa también se ha aplicado desde su creación en el desarrollo de videojuegos de diversas formas; algunas de las más destacadas son: control de elementos móviles, evaluación de estrategias y clasificación [14].

El primer caso, uso de la lógica difusa para el control de elementos móviles, ha permitido modelar con éxito el movimiento de vehículos en los videojuegos para conseguir movimientos mucho más realistas. Mediante esta técnica se consiguen movimientos suavizados, mucho más cercanos a los reales, y se evitan cambios bruscos en la dirección o velocidad de movimiento. Además del movimiento de vehículos, la lógica difusa también se ha empleado en el desarrollo de videojuegos para la animación realista de humanoides, cuyos elementos corporales están estructurados en un esqueleto.

Estas técnicas se pueden emplear a nivel individual, o bien, a nivel grupal, con el objetivo de animar grupos de objetos en movimiento que se dirigen hacia un objetivo común. Los integrantes del grupo pueden variar sus movimientos y conductas uno con respecto de otros sin alejarse del objetivo que tienen en común. De esta forma dejan de comportarse como un bloque que actúa exactamente de la misma forma y se otorga mayor realismo a la escena.

En el segundo caso, evaluación y determinación de estrategias mediante el uso de la lógica difusa, se consiguen comportamientos "más humanos". A la hora de modelar el comportamiento inteligente de los adversarios a los que se enfrenta un jugador, se puede tener en cuenta la posibilidad de que éste no cuenta con toda la información sobre el personaje principal, o bien, la información de la que dispone es incompleta. Esto hace también que el comportamiento de los adversarios sea mucho menos previsible y rompe la monotonía del videojuego.

Por último, la lógica difusa se emplea con mucha frecuencia en procesos de clasificación. A partir de ciertos parámetros como pueden ser: fuerza del personaje, potencia del arma, puntos débiles, etc, se puede realizar una clasificación para la obtención de un ranking que luego puede ser empleado en otros procesos relacionados con la Inteligencia Artificial del juego; como por ejemplo, en la elaboración de estrategias.

Fundamentos de la lógica difusa

En general, cualquier sistema difuso consta de cuatro partes fundamentales: (I) una primera parte de modelado en el que se define el conjunto de variables de entrada y sus dominios de definición, (II) proceso de fuzzificación en el que se convierten las entradas crisp o valores numéricos del sistema en entradas difusas, (III) motor de inferencia para el proceso de razonamiento y, finalmente, (IV) proceso de defuzzificación en el que se realiza la conversión inversa al punto (I) (ver Figura 28.19). Este último paso no siempre es necesario y se puede mantener una salida difusa si así se desea.

Fase de modelado

La fase de modelado consiste en especificar el conjunto de variables de entrada que utilizará posteriormente el sistema en el proceso de inferencia. Para cada una de las variables será necesario definir su dominio, constituido por el rango de valores numéricos que la variable puede tomar y los conjuntos difusos. Ejemplos de variables podrían ser: altura, peso, velocidad, presión, etc. Los conjuntos difusos abarcan un rango de valores y se les asocia una etiqueta lingüística. Por ejemplo, para la variable altura los conjuntos podrían ser: muy bajo, bajo, medio, alto y muy alto. Los conjuntos pueden adoptar diferentes formas: curvilíneas (Figura 28.18), triangulares (Figura 28.16) o trapezoidales (Figura 28.17) entre otras. En función de las características del problema a modelar será conveniente elegir una forma u otra.

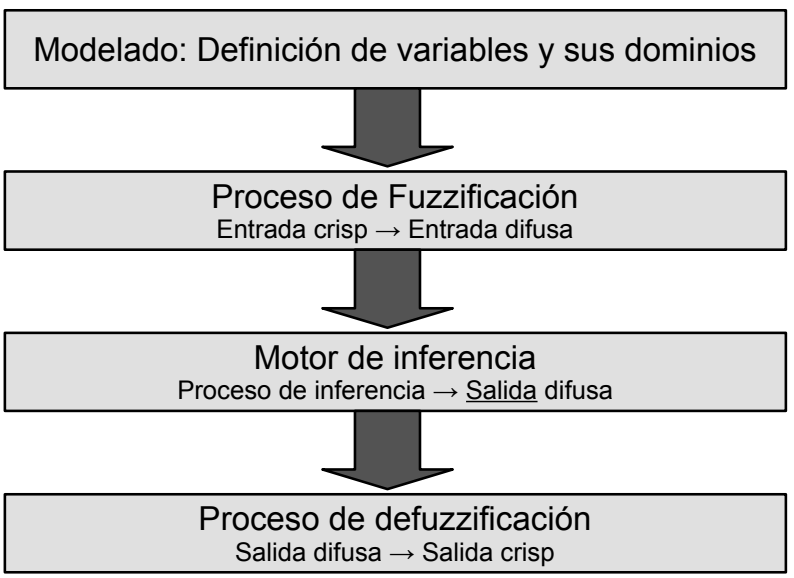


Figura 28.19: Esquema general de un sistema difuso.

En la Figura 28.20 se puede observar un ejemplo en el que se define un espacio difuso para la variable altura. Cada uno de los conjuntos difusos se define mediante un trapezoide y tiene asociado una etiqueta lingüística. Para definir cada uno de los trapecios y sus límites son necesarios cuatro valores numéricos. Tal como se puede apreciar, el valor de la altura varía desde 1.40 hasta 2.20 y todos los casos son abarcados por los conjuntos difusos definidos.

Los dominios de una variable se pueden variar fácilmente modificando los límites de los conjuntos; esta característica dota a los sistemas difusos de gran flexibilidad. El ejemplo que se muestra en la Figura 28.20 podría servir para clasificar los estudiantes de un curso en una universidad española. Suponga que quisiéramos clasificar los alumnos de una universidad noruega en la que la mayoría de alumnos supera la medida de 1.80 metros. En este caso, una persona que mida 1.80 puede ser considerado en ese contexto con una estatura media y no alta. Para tal caso, bastaría con modificar los límites de los conjuntos sin la necesidad de alterar el resto de partes del sistema.

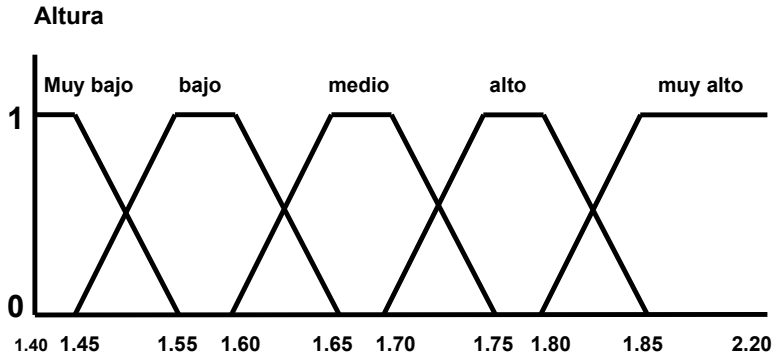


Figura 28.20: Dominio de definición o espacio difuso definido para la variable altura.

Una vez definidas las variables y sus dominios de definición, el siguiente paso consiste en la fuzzificación de variables a medida que reciban valores de entrada. Este paso consiste básicamente en el estudio de los valores de pertenencia a los conjuntos difusos en función de los valores *crisp* de entrada. Por ejemplo, supongamos que una persona mide 1.64, es decir, altura = 1.64. Dicho valor de altura estaría comprendido entre los conjuntos *bajo* y *medio*, con una pertenencia mayor al conjunto *medio*.

Un espacio difuso está correctamente definido, si la suma de todas las pertenencias a los conjuntos siempre es 1, independientemente del valor de entrada. Formalmente, la definición de la función de pertenencia asociada a un trapecio o conjunto difuso trapezoidal es de la siguiente manera:

$$\prod(u;a,b,c,d) = \begin{cases} 0 & u < a \\ \frac{(u-a)}{(b-a)} & a \leq u < b \\ 1 & b \leq u \leq c \\ \frac{(d-u)}{(d-c)} & c < u \leq d \\ 0 & u > d \end{cases} \tag{28.2}$$

donde u representa el valor de entrada de la función, y a, b, c y d los cuatro parámetros a partir de los cuales se define el trapecio, siendo 1 la altura máxima. Cada DDV_i (dominio de definición de la variable v_i) debe verificar las siguientes propiedades:

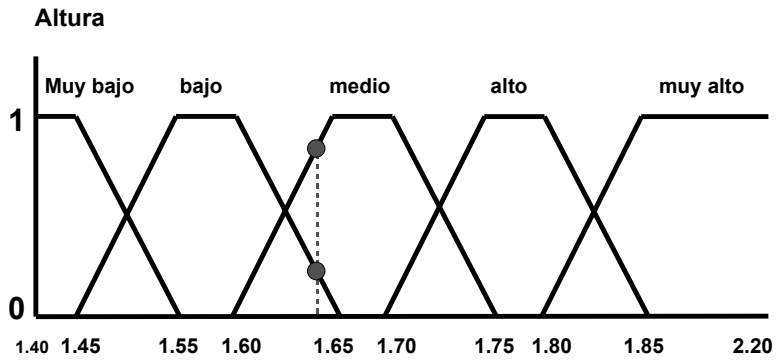


Figura 28.21: Estudio de los valores de pertenencia a los conjuntos difusos a partir de una altura de 1.64.

1. $\forall L_x \in DDV_i$, $altura(L_x) = 1$. L_x representa la etiqueta lingüística del conjunto x.
2. $\forall L_x, L_y \in DDV_i$, $nucleo(L_x) \cap nucleo(L_y) = \emptyset$. Pertenecen al núcleo de un conjunto difuso, representado mediante una etiqueta lingüística L, aquellos valores $x \in X_i$ que maximizan la función de pertenencia μ_L, es decir, $\mu_L(x) = 1$.
3. $\forall x \in X_i, \sum_{j=1}^{|DDV_i|} \mu_{L_j}(x) = 1$, siendo X_i el dominio donde se define v_i.

A continuación se muestra una posible implementación de las funciones que estudian el grado de pertenencia a conjuntos trapezoidales y triangulares:

Listado 28.5: Grado de pertenencia a un conjunto trapezoidal

```
1  double  FuzzyTrapezoid(double u, double a, double b, double c,
       double d)
2
3  {
4      double result = 0;
5
6      if (a <= u && u < b )
7          result = (u-a)/(b-a);
8      else if (b <= u && u <= c)
9          result = 1;
10     else if (c < u && u <= d)
11         result = (d-u)/(d-c);
12
13     return result;
14
15 }
```

Listado 28.6: Grado de pertenencia a un conjunto triangular

```
1  double  FuzzyTriangle(double u, double a, double b, double c)
2  {
3      double result = 0;
4
5
6      if (a <= u && u < b )
7          result = (u-a)/(b-a);
```

```
8       else if (u == b)
9           result = 1;
10      else if (b < u && u <= c)
11          result = (c-u)/(c-b);
12
13      return result;
14  }
```

Motor de inferencia

Una vez que se ha realizado el proceso de fuzzificación, para determinar el grado de pertenencia de las variables a los conjuntos difusos, comienza el proceso de inferencia o razonamiento para la toma de decisiones. Uno de los motores de inferencia más comunes es el basado en reglas difusas de tipo SI-ENTONCES. Este tipo de reglas está formado por un conjunto de antecedentes conectados mediante operadores lógicos (AND, OR, NOT) y uno o varios consecuentes. Cada antecedente a su vez consiste en una variable difusa y el rango de etiquetas lingüísticas que ésta debería tomar para que se produzca la activación de la regla. Veamos un ejemplo:

SI distancia {muy corta, corta} y fuerza_enemigo es {baja, muy baja, media}
ENTONCES *Ataque {agresivo}*

Para que se active la regla anterior la distancia entre el enemigo y el personaje principal debe tener un grado de pertenencia superior a cero en los conjuntos, *corta* o *muy corta*; lo mismo sucede con la fuerza del enemigo, en donde al menos se debe dar uno de los tres casos siguientes: *baja, muy baja* o *media*. En caso afirmativo, la regla se activa y el consecuente es *Atacar* con un determinado grado de pertenencia a *ataque agresivo*.

Como se puede apreciar una regla difusa abarca múltiples casos y, estos casos a su vez, tienen en cuenta un amplio rango de valores para las variables de entrada. Según Timothy Masters en su libro *Practical Neural Networks Recipes in C++* [61], un sistema formado por reglas difusas necesita entre un 50 % y un 80 % de reglas menos que necesitaría un sistema tradicional, para realizar las mismas tareas; por tanto, el mantenimiento es mucho más sencillo.

Evaluación de las reglas difusas

Para evaluar el grado de activación de una regla difusa, primero es necesario establecer como se resolverán los operadores lógicos empleados en los antecedentes. Existen diversas formas, pero una de las más comunes es emplear el máximo para el operador OR, el mínimo para AND y 1-valor de pertenencia, para el operador NOT. Si se aplican estos criterios, el grado de activación de una regla viene determinado por el mínimo de los máximos.

Si tenemos en cuenta la regla anterior, supongamos que el grado de pertenencia a los conjuntos *muy corta y corta* de la variable *distancia* es 0.2 y 0.8 respectivamente; por otro lado, supongamos que en el caso de la fuerza del enemigo el grado de pertenencia a muy baja es 1 y al resto es cero. La forma de determinar el grado de activación es el siguiente:

- distancia: MAX (0.2 y 0.8) = 0.8
- fuerza del enemigo MAX (1, 0, 0) = 1

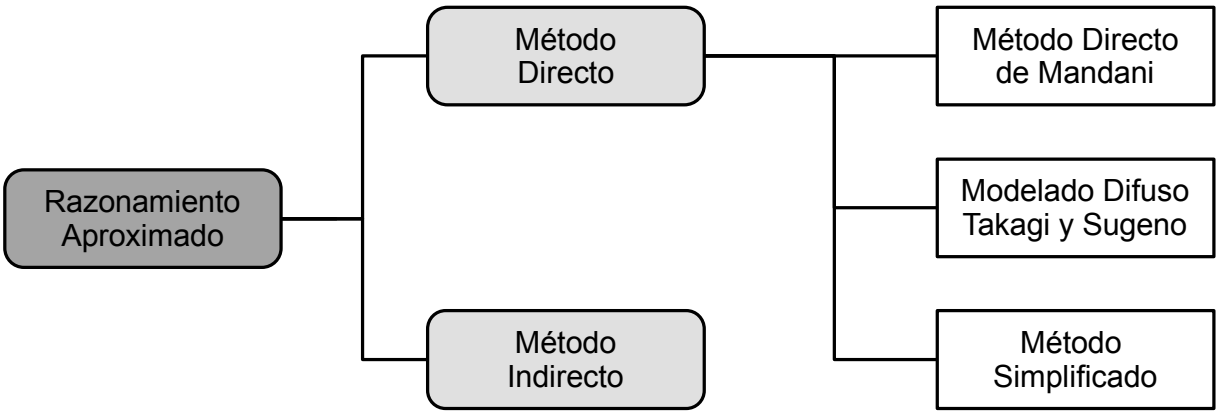


Figura 28.22: Clasificación de los Métodos de Razonamiento Aproximado.

- Tanto la distancia como la fuerza del enemigo están conectadas mediante un operador AND, por tanto: MIN(0.8, 1) = 0.8, determina el grado de activación de la regla.

Listado 28.7: Funciones asociadas a los operadores lógicos

```
1  double FuzzyAND(double A, double B)
2  {
3      return MIN(A, B);
4  }
5
6  double FuzzyOR(double A, double B)
7  {
8      return MAX(A, B);
9  }
10
11 double FuzzyNOT(double A) {
12     return 1.0 - A;
13 }
```

Veamos con mayor detalle uno de los métodos de inferencia más populares en la lógica difusa: Método de Mandani.

Proceso de razonamiento mediante el método de Mandani

Según Tanaka [96], los sistemas de razonamiento aproximado se pueden clasificar como se muestra en la figura 28.22.

Uno de los métodos más utilizado es el Método Directo de Mandani. Veamos un ejemplo de funcionamiento con dos reglas:

Regla 1: IF x es A_1 e y es B_1 THEN z es C_1
Regla 2: IF x es A_2 e y es B_2 THEN z es C_2

Donde A_1, A_2, B_1, B_2, C_1 y C_2 son conjuntos difusos. La figura 28.23 muestra el proceso de razonamiento que se expondrá a continuación. Supongamos x_0 e y_0 como entradas para las variables x e y de las premisas (parte antecedente de la regla). Denotaremos la entrada como (x_0, y_0).

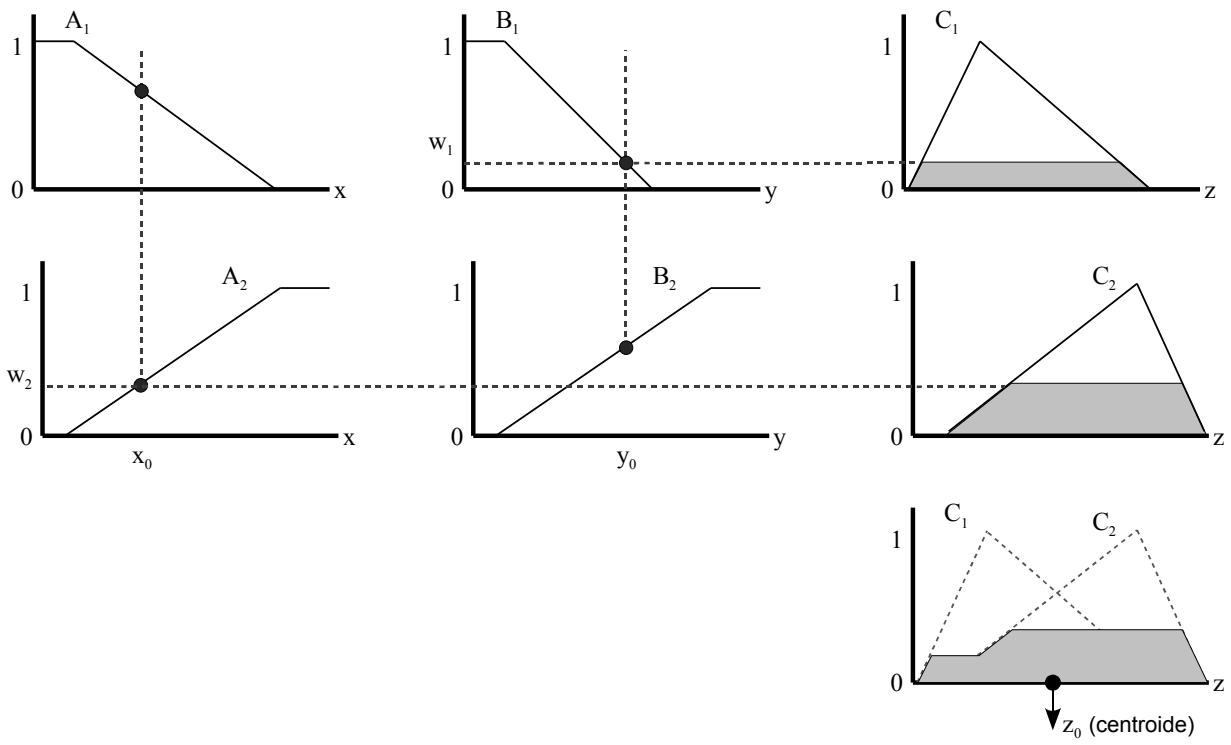


Figura 28.23: Proceso de razonamiento mediante el método de Mandani.

El proceso de razonamiento para esta entrada será el siguiente:

- **Paso 1:** Obtener el grado en que se satisfacen los antecedentes. En cada regla hay dos proposiciones en el antecedente. Para calcular el grado de satisfacción de cada regla se utilizarán, en este caso concreto, los operadores estándar sobre conjuntos difusos. En este ejemplo se emplearán los operadores estándar para T-normas (intersección de conjuntos - operador AND).

$$\textit{Regla 1}\text{: } W_1 = \text{mín}\left[\mu_{A_1}(x_0), \mu_{B_1}(y_0)\right]$$
$$\textit{Regla 2}\text{: } W_2 = \text{mín}\left[\mu_{A_2}(x_0), \mu_{B_2}(y_0)\right]$$

- **Paso 2:** Aplicar los resultados obtenidos en el paso anterior a los conjuntos difusos en la parte del consecuente, para así obtener la conclusión de cada regla.

$$\textit{Conclusión de la Regla 1}\text{: } \mu_{C_1}(x_0) = \text{mín}\left[W_1, \mu_{C_1}(z)\right] \quad \forall z \in Z$$
$$\textit{Conclusión de la Regla 2}\text{: } \mu_{C_2}(x_0) = \text{mín}\left[W_2, \mu_{C_2}(z)\right] \quad \forall z \in Z$$

- **Paso 3:** Construir la conclusión final mediante la agregación de todas las conclusiones parciales. En este caso, se ha empleado el operador T-conorma estándar (Unión de conjuntos - operador OR).

$$\textit{Conclusión final}\text{: } \mu_C(z) = \text{máx}\left[\mu_{C_1}(z), \mu_{C_2}(z)\right]$$

Proceso de defuzzificación

El proceso de defuzzificación es necesario sólo si se desea un valor real o *crisp* de salida en nuestro sistema (valor z_0 en la figura 28.23). El proceso más sencillo y ampliamente utilizado se conoce como método de *centro de gravedad*, en el que se obtiene el punto central del área generada en el consecuente de la regla.

Si tenemos en cuenta de nuevo el ejemplo anterior, donde se plantea el método de razonamiento de Mandani, los métodos de defuzzificación más comunes son:

- Tomar el **centro de masas** del conjunto difuso conclusión:

$$z_0 = \frac{\int \mu_c(z) z \, dz}{\int \mu_c(z) \, dz}$$

- Tomar el valor de la **máxima pertenencia** del conjunto difuso conclusión:

$$z_0 = \left(\max_z \mu_c(z) \right)$$

Caso de estudio: selección de estrategia de combate

En esta sección se plantea un ejemplo sencillo en el que se puede emplear un sistema difuso para determinar el comportamientos de los personajes que participan en un campo de batalla. Para no complicar el sistema de navegación de los personajes, podemos suponer que el campo de batalla es abierto, es decir, sin obstáculos. Dicho campo se puede modelar fácilmente con el uso de una matriz bidimensional, donde la fila y la columna en la que se encuentra un personaje determina su posición actual.

Inicialmente el frente enemigo se encuentra situado en un extremo (ya sea la primera fila o la primera columna) y el bando en el que nuestro personaje se encuentra en el extremo opuesto. El personaje que el usuario maneja cuenta con el apoyo de otros personajes aliados; el sistema difuso debe permitir modelar el comportamiento tanto de los enemigos como de los aliados.

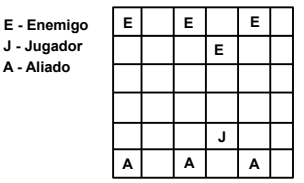


Figura 28.24: Campo de batalla representado como una matriz bidimensional. Elementos participantes: Enemigos, aliados y el propio jugador.

Con el objetivo de no introducir una excesiva complejidad en el problema, supongamos que la estrategia que puede adoptar cada personaje depende de un número reducido de variables:

- Energía del objetivo y del personaje que valora la estrategia
- Distancia entre ambos
- Grado de ocupación, es decir, si está libre o, por el contrario, se encuentra envuelto en una batalla con una o varias entidades.

Para dichas variables se pueden definir los siguientes conjuntos difusos:

- Energía: baja, muy baja, media, alta y muy alta.
- Distancia: muy cerca, cerca, distancia media, lejos y muy lejos
- Grado de ocupación: libre, muy poco ocupado, poco ocupado, ocupado, muy ocupado.

Tal como se vio en las las secciones anteriores, existen diferentes alternativas para definir los conjuntos difusos correspondientes a las etiquetas lingüísticas especificadas en el listado anterior: trapezoidales, triangulares, mediante curvas, etc. Queda a elección del lector, elegir una de las opciones en función del comportamiento del sistema deseado.

Por otro lado, las variables de salida son las que se incluirán en el consecuente de las reglas y determinarán las acciones básicas de cada individuo: acercarse, huir, atacar y defenderse. A continuación se detalla una breve descripción sobre cada una de ellas:

- **Atacar**: Si un enemigo ataca y el otro no se encuentra en posición de defensa, recibirá el 100 % de daño producido por el ataque. A medida que la energía del individuo disminuya, el ataque debería producir un daño menor. Por el contrario, si el individuo se encuentra en posición de defensa, podría evitar el daño producido por el ataque (no siempre).
- **Defenderse**: Esta acción permite al individuo que la realiza recuperar parte de su energía. Además, podría evitar daños en el caso de recibir un ataque.
- **Huir**: Desplazarse una casilla en el tablero incrementando la distancia con respecto al enemigo.
- **Acercarse**: Desplazarse una casilla en el tablero disminuyendo la distancia con respecto al enemigo.

Una vez especificadas las variables de entrada/salida y definido el dominio difuso para cada una de ellas, el siguiente paso consiste en construir una base de conocimiento formada por reglas difusas que van a determinar las conductas. En función de la estrategia que queramos adoptar, se debe dar preferencia en el antecedente de las reglas a unas variables u otras. Por ejemplo, podríamos optar por atacar siempre a aquellos enemigos que tengan una energía más baja o bien, podríamos optar por atacar al más cercano. Algunos ejemplos de reglas difusas podrían ser los siguientes:

- SI Energia_Enmigo{muy baja, baja} y Ocupacion {baja} y Mi_energia{media, alta, muy alta} y Distancia{muy cerca} ENTONCES *Atacar*
- SI Mi_Energia{muy baja, baja} y Distancia{muy cerca} ENTONCES *Defenderse*

Si finalmente optamos por realizar una de las cuatro acciones posibles en cada turno: acercarse, huir, atacar o defenderse; tan sólo sería necesario estudiar la activación de las reglas y actuar según aquella que tenga un grado de pertenencia mayor.

Una vez implementando el sistema difuso, sería interesante realizar modificaciones del dominio de definición de cada variable para poder observar como varía el comportamiento de los personajes sin la necesidad de variar la lógica de programa. La inclusión de nuevas reglas o modificación de las existentes, también contribuye al cambio de conductas.

28.2.2. Algoritmos genéticos

Los algoritmos genéticos (AG) están basados en la propia naturaleza, en concreto, en la teoría de la evolución en donde los seres vivos evolucionan de generación en generación adaptándose al entorno que habitan. Los geneticistas y biólogos evolutivos afirman que la evolución no optimiza necesariamente, sino que más bien se adapta y optimiza localmente en el espacio y el tiempo; es decir, la evolución no necesariamente significa progreso y los individuos de una generación no tienen porque ser mejores a los de generaciones anteriores, simplemente, se **adaptan** mejor al entorno actual.

Imaginemos por un momento, que los alimentos se encontraran únicamente en lo alto de árboles de una altura elevada. Las jirafas, animales de cuello muy largo, u otros animales con la capacidad de trepar desarrollada, serían los únicos que tendrían acceso a estos alimentos y, por tanto, los únicos que podrían sobrevivir. Este hecho no hace a las jirafas o animales trepadores mejores que los de otra especie, simplemente, los convierte en seres mejores adaptados al medio.

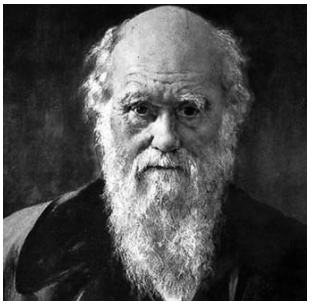


Figura 28.25: *Charles Darwin* (12 de febrero de 1809 – 19 de abril de 1882), propulsor de la teoría de la evolución mediante selección natural.

En los AG existe un espacio (normalmente amplio) de soluciones candidatas al problema que se está tratando (conocidas también como individuos). Esta población tiende a **evolucionar** hacia mejores soluciones mediante un proceso iterativo. Cada una de estas soluciones se puede entender como un cromosoma y, estos cromosomas, a su vez están constituidos por una cadena de genes. Estos genes representan la configuración genética del individuo o solución, y cada uno de ellos es una parte importante en la solución global; dependiendo de como estén combinados se obtiene soluciones más o menos óptimas. De forma general, el proceso evolutivo y de adaptación consiste en una *selección* de los mejores individuos de cada generación (de esta forma se orienta la búsqueda a las áreas más prometedoras); así como el *cruce* entre éstos o la *mutación* para dar lugar a los individuos de la siguiente generación, más evolucionados que los anteriores.

Adaptabilidad

Es la principal característica de los algoritmos genéticos; permiten encontrar soluciones que se adecuan a un entorno.

La evolución es una forma de adaptación más potente que el simple aprendizaje. En el desarrollo de videojuegos puede ser una herramienta realmente útil si se desea modelar entornos dinámicos y generar soluciones que se adapten a estos cambios. De esta forma, es posible desarrollar vídeojuegos mucho más realistas en donde las situaciones y comportamientos dejan de ser monótonos y previsibles; se producen un amplio abanico de posibilidades y el entorno de manera global se adapta a estas situaciones.

Operadores evolutivos

Los operadores evolutivos básicos son la selección, cruce y mutación.

Los AG son también realmente apropiados para conseguir configuraciones óptimas. Imagínese, por ejemplo, el juego *Gran Turismo 4* en el que para cada circuito existen coches y características mecánicas que se adaptan mejor para lo obtención de mejores tiempos. Si se tiene en cuenta que existen más de 500 coches y docenas de características configurables, encontrar una configuración óptima es realmente complicado. Un AG podría simular múltiples configuraciones y calcular el beneficio proporcionado por cada una de ellas.

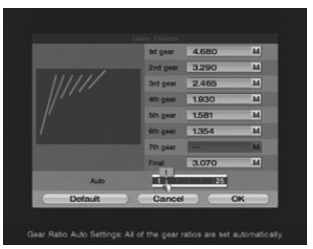


Figura 28.26: Configuración de la relación de marchas en el juego GT4

Esquema general de un algoritmo genético

Como se comentó anteriormente, los AG constan de un conjunto de procedimientos de búsqueda adaptativos. El primer requisito para un AG es la **constitución de una población inicial** formada por un conjunto de cromosomas. La generación de dicha población se puede producir de forma absolutamente aleatoria, o bien, combinándola con posibles soluciones candidatas que sean conocidas. Independientemente del método de generación, la población inicial debe poseer diversidad estructural para tener una representación de la mayor parte de la población y no caer en una convergencia prematura. La formación de la primera generación es vital para llegar con éxito a soluciones prometedoras en las futuras generaciones.

Diversidad Estructural

En un algoritmo genético es fundamental que la población inicial o primera generación posea diversidad estructural para converger a soluciones adecuadas

Una vez que se ha constituido la primera población o primera generación, se procede a la ejecución de un proceso iterativo. Este proceso se basa en el estudio de la calidad de cada una de las soluciones mediante el uso de una *función de aptitud*; es decir, esta función informará sobre lo "buena" que es una solución. Después de estudiar la aptitud de cada solución o cromosoma, comienza un **proceso de selección**, de tal forma que los cromosomas elegidos serán combinados para dar lugar a los individuos de la siguiente generación. Los cromosomas con mejor aptitud tienen siempre más probabilidades de ser seleccionados.

Función de aptitud

Permite evaluar la bondad de una solución

Entre los cromosomas elegidos se aplican **operadores genéticos** - reproducción para la generación de nuevos individuos: mutación, cruce, evaluación, etc. Finalmente, el AG debe finalizar el proceso iterativo cuando se alcance la solución óptima. En caso de que no sea posible obtener la mejor, se pueden aplicar otros criterios de parada: i) determinar al comienzo un número máximo de iteraciones y detener el procedimiento cuando se alcance esa iteración, ii) finalizar cuando no se produzcan cambios en la población o éstos sean mínimos.

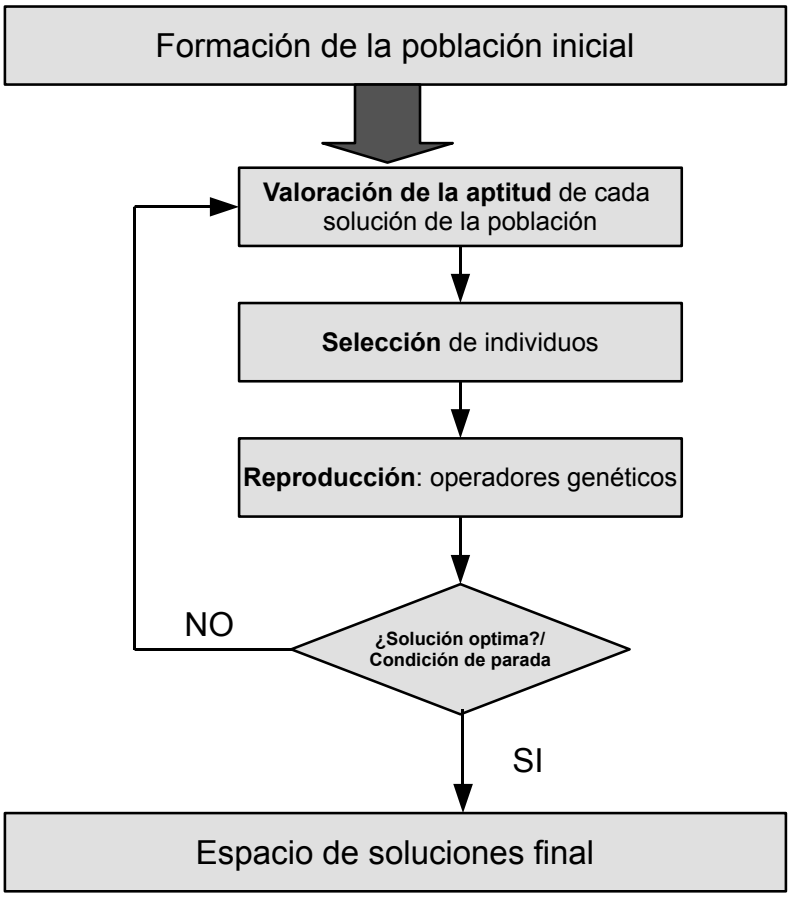


Figura 28.27: Esquema general de un algoritmo genético.

Representación del problema

Antes de constituir la población inicial y comenzar el proceso iterativo descrito en la sección anterior, es fundamental determinar la forma en la que se van a representar las soluciones. Cuando se elige o diseña un modo de representación se está definiendo también el espacio de estados en el que el AG va a realizar la búsqueda de soluciones. Por tanto, es fundamental ser cuidadoso a la hora de definir la estructura genética ya que de ello depende en gran medida el éxito o fracaso del AG en la búsqueda de soluciones óptimas, así como la eficiencia y coste computacional invertido.

Una forma de representación común es el uso de cadenas binarias, formadas por 1's y 0's, en donde cada dígito representa el valor de algún aspecto de la solución. Cuantos más bits tengamos en la cadena, más posibles estados podremos definir (2^n). En función del número de estados que queramos representar, se emplearán cadenas binarias de mayor o menor longitud. Por ejemplo, un juego clásico de *comecocos* en el que el personaje principal tiene cuatro posibles opciones en cada turno: movimiento hacia arriba, abajo, izquierda o derecha. Son cuatro estados los que podemos representar con dos bits:

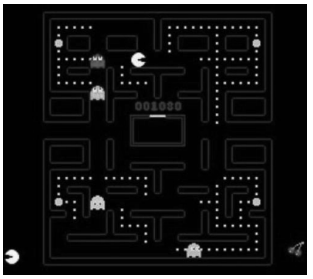


Figura 28.28: Juego clásico de comecocos en el que los cuatro movimientos o estados posibles del personaje principal son: movimiento hacia arriba, abajo, izquierda y derecha

- 00: mover arriba
- 01: mover abajo
- 10: mover izquierda
- 11: mover derecha

Cuando el personaje principal se encuentra en alguno de los estados, la mutación o cambio en alguna de las partes de la estructura genética produce un cambio de estado. La función de aptitud determinará cuál es el mejor de los cambios de estado teniendo en cuenta que tras el movimiento, el personaje puede encontrarse con un muro, una casilla libre, un fantasma al que puede comer, o bien, un fantasma que podría eliminarlo.

En algunas ocasiones el modo de representación binario es limitado y no permite afrontar con garantías el problema a tratar. En este caso, se puede optar por emplear valores numéricos en las cadenas; así se permite una mayor precisión y complejidad. Una segunda opción es emplear cadenas de letras, en donde cada una de las letras representa un aspecto específico de la solución.

Los tres métodos anteriores facilitan el uso de operadores genéticos que alteren las cadenas de genes: el primero de ellos intercambiando los ceros por unos y viceversa; el segundo de ellos incrementando o decrementando los valores numéricos con una cantidad adicional; finalmente, el último método de representación permite el intercambio de letras.

Binaria	1	0	1	1	0	0	1
Enteros	7	3	3	5	7	1	2
Reales	2.1	3.4	5.6	4.5	3.2	9.1	8.8
Caracteres	A	C	A	A	B	C	A

Figura 28.29: Modos de representación para secuencias genéticas

Operadores Genéticos

Los operadores genéticos se pueden clasificar en tres grandes clases: *selección*, *mutación* y *cruce*. En cada una de estas categorías existen múltiples variantes y se pueden emplear y combinar varias de ellas en un AG.

Operadores de selección

- **Selección elitista**: se escogen los miembros más aptos en cada generación. Este método se puede entender como un proceso de clonación, en el que se copian directamente los mejores miembros de una generación a la siguiente, de tal forma que se garantiza que no se pierdan en el proceso de búsqueda. Sin embargo, abusar de este método no es conveniente ya que la búsqueda puede converger a soluciones que representan máximos locales y podrían evitar la generación de una solución que represente un máximo global.
- **Selección proporcional a la aptitud**: Los individuos más aptos tienen mayor probabilidad de ser seleccionados pero no la certeza. Esta probabilidad se podría calcular en función del valor de aptitud y de la suma totales de aptitudes: $p_i = f_i / \sum_j f_j$. Para determinar si un cromosoma se selecciona para una posterior reproducción, se puede generar un número aleatorio entre 0 y 1; si el valor generado es inferior a p_i, entonces el individuo es seleccionado.
- **Selección por rueda de ruleta**: En este caso, el individuo también tiene una probabilidad de ser seleccionado. Sin embargo, p_i es proporcional a la diferencia entre su aptitud y la del resto de soluciones. Se puede entender como una ruleta en la que cada sector representa la probabilidad de cada individuo para ser seleccionado. Cuanto mayor sea la probabilidad, mayor será el sector para ese individuo. Cada vez que se gira la ruleta y se detiene en alguno de los sectores se realiza la selección de uno de los miembros. A medida que se obtengan nuevas generaciones, las soluciones deberían ir convergiendo hacia un punto en común de tal forma que la diferencia de aptitud entre los individuos debe ser menor y la probabilidad de ser seleccionados muy similar.
- **Selección escalada**: este método soluciona el problema planteado en el punto anterior, en el que los miembros de una población poseen valores de aptitud similares. Esto suele suceder en generaciones posteriores y no es conveniente hasta entonces utilizarlo. Con este tipo de selección se permite que la función de aptitud sea más discriminatoria.
- **Selección por torneo**: En este método se eligen subgrupos de la población y se enfrentan unos con otros. El vencedor de la competición es seleccionado para el proceso de reproducción.
- **Selección por rango**: En este método se realiza un *ranking* de individuos en base a un rango numérico asignado a cada uno de ellos en función de la aptitud. La ventaja principal de este método es que evita que individuos con un grado de aptitud demasiado elevado ganen dominancia sobre los que tienen un valor más bajo. Esto es crítico sobre todo al principio, ya que soluciones que podrían ser descartadas en las primeras generaciones podrían evolucionar progresivamente hasta soluciones óptimas globales.
- **Selección generacional**: Solo pasa a la siguiente generación la descendencia de la generación anterior. En ningún momento se copian individuos.
- **Selección por estado estacionario**: En las primeras generaciones se realiza una selección menos compleja, poco rigurosa y menos discriminatoria, mientras que en las sucesivas ocurre justo lo contrario. De esta forma se reduce el tiempo total de cálculo en la búsqueda de soluciones.

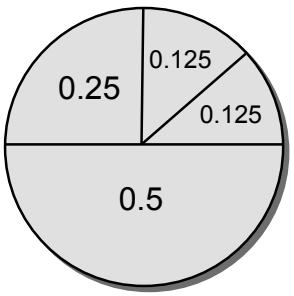


Figura 28.30: Selección genética mediante el método de ruleta. Cada sector representar la probabilidad de un individuo de ser seleccionado

Mutación

La mutación consiste en alterar de forma aleatoria una o varias partes de la solución. En el caso de la naturaleza las mutaciones pueden llegar a ser letales, sin embargo, contribuyen a la diversidad genética de la especie. No es conveniente utilizar con una frecuencia alta este operador para no reducir el AG a una búsqueda aleatoria; tan sólo es aconsejable cuando el algoritmo esté estancado. Una buena frecuencia de mutación es 1/100 o 1/1000.

Para cada uno de los genes que constituyen una solución se genera un número aleatorio, entre 0 y 1; si ese número es inferior a 0.01 o 0.001 se realiza el intercambio del valor.

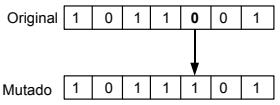


Figura 28.31: Ejemplo de mutación en una cadena binaria

Cruce (Crossover)

Este es el operador principal en un AG y el que se utiliza con mayor frecuencia. Consiste en el intercambio genético entre dos cromosomas; para ello se escogen dos de los miembros elegidos en el proceso de selección y se intercambian algunas de las partes de la cadena formada por los genes. Existen diferentes tipos de operadores de cruce:

- **Cruce de 1 punto**: Se divide la cadena de genes en un punto en concreto y se intercambian entre los dos miembros. En la figura 28.32 se puede ver un ejemplo.
- **Cruce de n-puntos**: Igual que el caso anterior, pero las cadenas se dividen n veces en puntos diferentes (ver figura 28.33).
- **Cruce uniforme**: Se genera un patrón y se realiza el intercambio de acuerdo a ese patrón. Por ejemplo, en el caso de la imagen 28.34 se realizan varios cortes en diferentes puntos de la cadena y sólo se intercambian aquellas que se encuentran en posiciones impares. En el caso de emplear cadenas binarias como método de representación, se puede generar un patrón fácilmente con ceros y unos obtenidos de forma aleatoria. Aquellos lugares en los que se encuentre un uno, representan los fragmentos de la cadena que deben ser intercambiados.
- **Cruce especializado**: El problema de los métodos de cruce anteriores es que pueden generar casos que representan soluciones no válidas. En este tipo de cruce se evalúa si un cambio genera una solución no válida, y en tal caso, no realiza el intercambio.

Además de elegir los operadores genéticos que se van a emplear hay que determinar también su frecuencia de uso. Dependiendo de la fase o etapa de la búsqueda en la que se encuentre un AG, es conveniente emplear unos tipos u otros. En un principio, los métodos más eficaces son la mutación y el cruce; posteriormente, cuando la población o gran parte de ésta converge el cruce no es demasiado útil ya que nos vamos a encontrar con individuos bastantes similares. Por otro lado, si se produce un estancamiento, y no se consiguen soluciones en las nuevas generaciones que mejoren el valor de aptitud, la mutación tampoco es aconsejable ya que se reduce el AG a una búsqueda aleatoria. En tal caso, es aconsejable utilizar otro tipo de operadores más especializados.

Procesamiento paralelo

Se exploran varias soluciones de forma simultánea

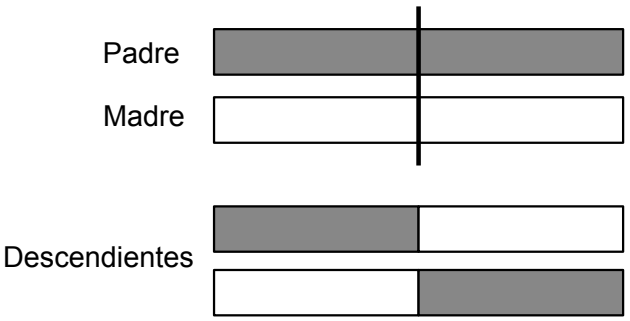


Figura 28.32: Cruce de un punto entre dos cromosomas.

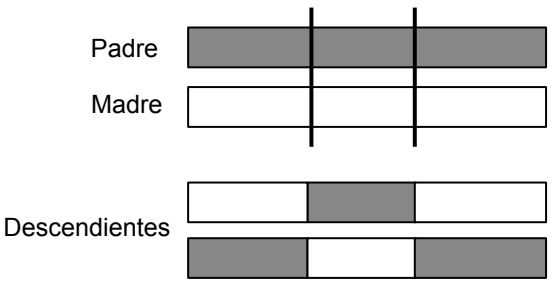


Figura 28.33: Cruce de n puntos entre dos cromosomas. En este caso n = 2.

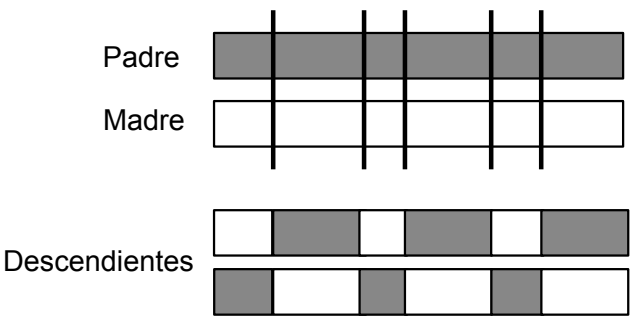


Figura 28.34: Cruce uniforme de dos cromosomas. En este caso n = 2.

Ventajas de los algoritmos genéticos

Una de las grandes ventajas de los AG es que estos tienen descendencia múltiple y pueden explorar el espacio de soluciones de forma paralela. A diferencia de los algoritmos que buscan una solución siguiendo un único camino, los AG no se ven realmente penalizados en el caso de que uno de los caminos no llegue a una solución satisfactoria. Gracias a esta ventaja, los AG tienen un gran potencial para resolver problemas cuyo espacio de soluciones es grande o para resolver problemas no lineales.

Otra ventaja importante es que los AG son apropiados para manejar múltiples parámetros simultáneamente. Es posible, que el AG no encuentre una solución óptima global y pueden existir varias soluciones al problema en cuestión y que algunas de ellas optimicen algunos de los parámetros de entrada y otras soluciones, optimicen otros distintos.

Múltiples parámetros

Manejo simultáneo de múltiples parámetros de entrada, hasta encontrar la configuración adecuada

Además, los AG no tienen porque conocer nada sobre el problema a resolver. De hecho, el espacio inicial de soluciones puede ser generado aleatoriamente en su totalidad, y que estos individuos converjan hacia soluciones válidas para el problema. Esta característica es muy importante y hace que los AG sean realmente útiles para resolver problemas en el que se posee un gran desconocimiento sobre las posibles soluciones y no sería posible definir una de ellas de forma analítica.

Desconocimiento

Permite llegar a las soluciones de forma no analítica en caso de que exista desconocimiento sobre como resolver el problema

Limitaciones de los algoritmos genéticos

Una de las posibles limitaciones de los AG reside en el modo de representación de las cadenas de genes. Como se comentó anteriormente, el método utilizado debe tolerar cambios aleatorios que no produzcan una gran cantidad de soluciones o cadenas que supongan resultados sin sentido. Normalmente, los tres métodos de representación expuestos con anterioridad (representación mediante cadenas binarias, cadenas formadas por números enteros o reales, cadenas de letras) ofrecen la posibilidad de selección, mutación y cruce.

Otro problema con el que nos encontramos en los AG es la definición de una función de aptitud apropiada. Si la definición no es correcta puede que el AG sea incapaz de encontrar las soluciones al problema. Además, de elegir una función de aptitud apropiada, es necesario también configurar el resto de parámetros del AG correctamente, como por ejemplo, tamaño inicial de la población, frecuencia de cruce, frecuencia de mutación, etc.

Si el conjunto inicial es demasiado pequeño, es posible que el AG no explore el espacio de soluciones correctamente; por otro lado, si el ritmo de cambio genético es demasiado alto o los procesos de selección no hacen su trabajo correctamente, puede que no se alcancen las soluciones apropiadas.

Representación

El modo de representación elegido debe facilitar la aplicación de los operadores genéticos

Cuando un individuo destaca los suficiente en un principio con respecto al resto de elementos de la población, se puede producir una convergencia prematura. El problema es que este individuo se puede reproducir tan abundantemente que merma la diversidad de la población. De esta forma el AG converge hacia un óptimo local demasiado pronto y se limita la posibilidad de encontrar óptimos globales. Esto suele suceder sobre todo en poblaciones pequeñas. Para solucionar este problema, los métodos de selección empleados deben reducir o limitar la ventaja de estos individuos; los métodos de selección por rango, escalada y torneo son aptos para solucionar este problema.

Función de aptitud

Es fundamental diseñar una función de aptitud adecuada para encontrar las soluciones deseadas

Por último, cabe destacar que no es muy aconsejable el uso de AG para problemas que se pueden resolver de forma analítica, sobre todo por el alto coste computacional que implican. El uso de AG es apropiado en problemas no lineales, donde el espacio de soluciones es grande y no se tiene una idea clara sobre la/las posibles soluciones. Si esa solución además depende de la configuración de múltiples parámetros que deben ser ajustados, los AG son idóneos para encontrarla.

Máximos locales

Se producen cuando ocurre una convergencia prematura por la dominancia de uno de los individuos

$$f(x) = x^2$$

Figura 28.35: Optimización de la función $f(x) = x^2$ en el intervalo [0,31] mediante el uso de algoritmos genéticos

Ejemplo de aplicación de un algoritmo genético: $f(x) = x^2$

En esta sección se describirá un ejemplo sencillo en el que se aplica un AG para maximizar la función $f(x) = x^2$ y donde x es una variable que toma valores numéricos en el intervalo [0,31] [87]. Lógicamente, este problema se puede resolver de forma analítica y no es necesario el uso de un AG. Sin embargo, el ejemplo ayudará a comprender con facilidad la forma en la que se aplica un AG.

Dado que x puede tomar valores entre 0 y 31, elegimos un modo de representación de cadenas binarias de 5 bits, donde 00000 representa el número 0 y 11111 el 31. En la tabla 28.1 se muestra la población inicial formada por 6 posibles soluciones (escogidas de forma aleatoria): 21, 8, 18, 19, 30 y 9. En la segunda columna se representan los cromosomas equivalentes a la representación binaria de estos números. Finalmente, en la cuarta columna se muestra el valor de aptitud de cada una de las soluciones, siendo el número 30 la que tiene un valor mayor de aptitud.

Índice	Cromosoma	Entero x	Aptitud $f(x)$
1	10101	21	441
2	00100	8	64
3	10010	18	18
4	10011	19	361
5	11110	30	900
6	01001	9	81
		Promedio	361
		Máximo	900

Cuadro 28.1: Población inicial de individuos

Una vez que el conjunto de soluciones inicial está disponible, debe comenzar el proceso de selección y generar los descendientes. En este ejemplo, se ha elegido un modo de selección por torneo, en el que los cromosomas se enfrentan por pares. La forma de organizar los enfrentamientos es aleatoria y, al menos, cada cromosoma debe participar en alguno de los torneos. En este caso, al tener 6 cromosomas, serán necesarios 6/2 = 3 enfrentamientos. En la tabla 28.2 se puede apreciar como en una primera ronda, de forma aleatoria, se han enfrentado los pares de cromosomas (2,5) , (1,6) y (3,4). Para mantener una población de seis individuos, se ha vuelto a realizar un segundo torneo (filas 3-6) en el que se han enfrentado los pares: (1,5) , (2,3) y (4,6). Después de la selección se aprecia un incremento notable de la aptitud media en los miembros de la generación, aunque se mantiene el valor máximo.

Índice	Torneos	Cromosoma	Entero x	Aptitud $f(x)$
1	2,5	11110	30	900
2	1,6	10101	21	441
3	3,4	10011	19	361
4	1,5	11110	30	900
5	2,3	10010	18	324
6	4,6	10011	19	361
			Promedio	547
			Máximo	900

Cuadro 28.2: Selección por torneo. Enfrentamiento entre los cromosomas del conjunto inicial

Realizada la selección se pueden aplicar operadores genéticos de cruce y mutación. En la tabla 28.3 se aplican operaciones de cruce en un sólo punto entre los cromosomas 1-2, 3-4 y 5-6. Tal como se puede apreciar los cruces no se realizan siempre en el mismo punto, en el primer caso se realiza en el cuarto punto, en el segundo punto para los cromosomas 3-4 y en el tercer punto para el par de cromosomas 5-6. Por otro lado, la mutación solo se produce en el último bit del cromosoma 4 y como se puede observar, en este caso, la mutación es dañina ya que disminuye la aptitud de la solución. En este último grupo la aptitud promedio disminuye pero finalmente se encuentra el máximo global y, por tanto, no es necesario continuar.

Caso de estudio: juego de las 8 reinas

El problema de las 8 reinas consiste en ubicar las reinas en un tablero de 8x8 de tal forma que no se ataquen entre sí. Dos reinas no se atacan si no se encuentran en la misma diagonal. Un AG es apropiado para este problema y puede ayudar fácilmente a determinar las diferentes combinaciones existentes en la ubicación de cada reina en el tablero. Para un tablero de 8x8 y 8 reinas existen 12 patrones o soluciones correctas posibles. En la tabla 28.4 se muestra uno de ellos.

Índice	Cromo.	Punto	Cruce	Muta.	Entero x	Aptitud f
1	1111|0	4	11111	11111	31	961
2	1010|1	4	10100	10100	20	400
3	10|011	2	10110	10110	22	484
4	11|110	2	11011	1101**0**	26	676
5	100|10	3	10011	10011	19	361
6	100|11	3	10010	10100	18	324
					Promedio	534
					Máximo	961

Cuadro 28.3: Operadores de cruce y mutación

		Q					
					Q		
	Q						
			Q				
Q							
			Q				
							Q
				Q			

Cuadro 28.4: Una de las soluciones al problema de las 8 reinas

Para representar las soluciones se pueden emplear cadenas de valores enteros con números pertenecientes al intervalo [1,8]. Cada solución o cromosoma será una tupla de 8 valores $(q_1, q_2, ..., q_8)$ en donde la posición que ocupa cada q_i en la tupla representa la columna y el número indicado en la tupla la fila. De esta forma, la solución que se muestra en la tabla 28.4 se representaría de la siguiente forma:

(5, 3, 1, 6, 8, 2, 4, 7)

La función de adaptación es la que nos debe indicar la bondad de una solución. En este caso una solución perfecta es aquella en la que no existe conflictos entre las reinas. En dicha función se podrían contabilizar los conflictos, es decir, cuantas reinas se encuentran en la misma diagonal. Por tanto, cuando menor sea el valor devuelto por esta función, mejor será la solución.

El procedimiento que se puede seguir para resolver el problema es el siguiente:

- Generación de una población inicial
- Evaluar la población. Obtener el valor de adaptación para cada una de las soluciones de la población.
- Generar una población a partir de la anterior. Para ello se pueden emplear varios operadores genéticos, por ejemplo, el elitista en el que copiamos a la siguiente generación algunas de las mejores soluciones de la generación actual. Selección por ruleta y torneo, y combinar las soluciones elegidas para generar descendencia mediante cruce y mutación.
- Volver al paso 2 hasta llegar a un número máximo de generaciones o hasta alcanzar el objetivo (cero conflictos entre las reinas).

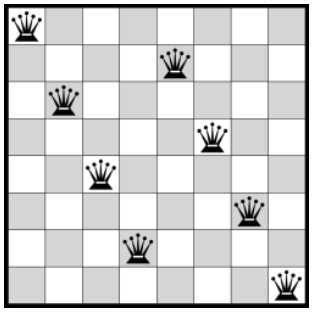


Figura 28.36: En este caso la función de adaptación devolvería el valor 1 debido al conflicto entre la reina situada en la esquina superior izquierda, y la reina situada en la esquina inferior derecha. La solución no sería óptima

Ventajas

Dos de las grandes ventajas que proporcionan las redes neuronales son: procesamiento en paralelo y conocimiento distribuido

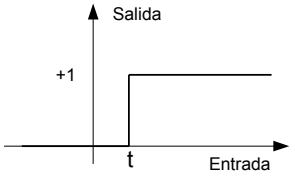


Figura 28.38: Función escalón

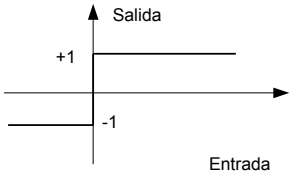


Figura 28.39: Función signo

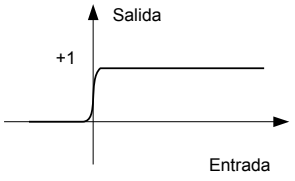


Figura 28.40: Función sigmoidea

28.2.3. Redes neuronales

Las redes neuronales son un paradigma de aprendizaje y procesamiento automático. Se basa en el modo de procesamiento del cerebro humano en el que existe multitud de neuronas interconectadas entre sí y colaboran para producir estímulos de salida. La neurona esta formada por el cuerpo o **soma**, en donde se encuentra el **núcleo**. Del cuerpo de la célula surgen diversas ramificaciones conocidas como **dendritas** y sale una fibra aún más larga denominada **axón**. La conexión entre neuronas se realiza a través del axón y las dendritas; a esta unión se le conoce como **sinapsis**. Las neuronas emiten señales y éstas se propagan a lo largo de la red gracias a reacciones electroquímicas. Las sinapsis liberan sustancias químicas transmisoras y entran en las dendritas, de tal forma que se eleva o se reduce el potencial eléctrico de la célula. Cada célula tiene un nivel o valor umbral de activación; cuando este se rebasa se envía al axón un impulso eléctrico que llega al resto de neuronas. Las sinapsis que aumentan el potencial son conocidas como **excitadoras** mientras que las que producen el efecto inverso son conocidas como **inhibidoras**.

Una red neuronal artificial está constituida por una serie de nodos conectados entre sí. A cada una de las conexiones se le asigna un peso numérico w_i que determina la influencia de dicha conexión en la propia neurona. Las redes neuronales artificiales representan mecanismos potentes para el procesamiento paralelo y distribuido; paralelo porque en la red existen varias neuronas al mismo tiempo procesando información y distribuido porque el conocimiento está distribuido a lo largo de toda la red y no focalizado en un punto concreto. En la fifura 28.41 se muestra el esquema general de una unidad mínima de procesamiento en una red neuronal.

Tal como se puede apreciar, la neurona recibe una o varias entradas. En cada una de las conexiones existe asociado un peso w_i que determina el grado de influencia de la entrada conectada a la neurona. En la propia neurona, existe una **función base** h que se encarga de combinar los valores de entrada teniendo en cuenta los pesos. Normalmente la función base es una suma ponderada $\sum_j w_j entrada_j$. Una vez combinados los valores de entrada se envía el valor calculado a la **función de activación** g. Dicha función determinará el valor de salida de la neurona. Existen multitud de funciones de activación, las más frecuentes son: *función escalón* (figura 28.38), *función signo* (figura 28.39) y *función sigmoidea* (figura 28.40).

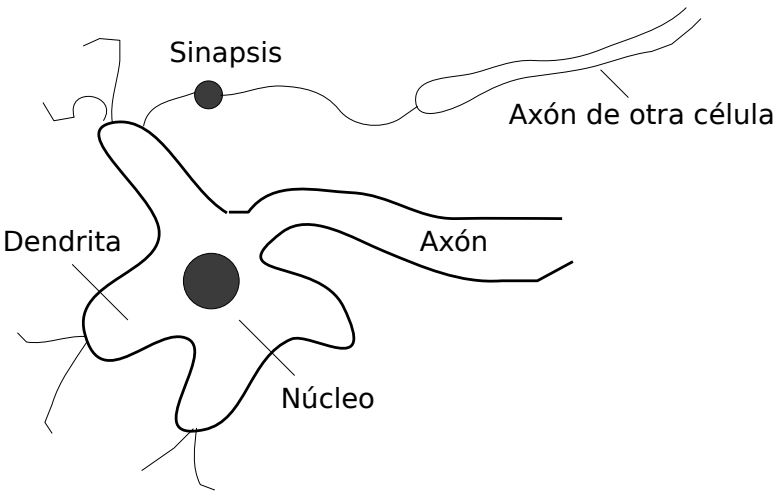


Figura 28.37: Partes de una célula nerviosa o neurona.

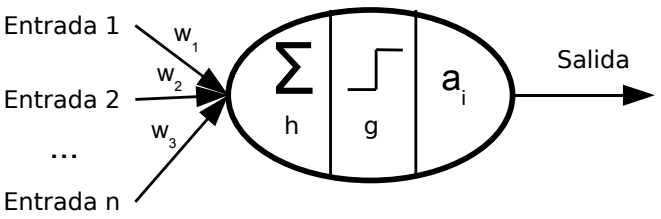


Figura 28.41: Unidad mínima de procesamiento en una red neuronal.

En la función de activación de escalón, se establece un valor umbral t. Si el valor numérico resultante de la combinación de las entradas con los pesos es superior a t entonces se activa la neurona pasando a valer su salida 1. En caso contrario sería cero.

$$escalon(x) = \begin{cases} 1 & \text{si } x \geq t \\ 0 & \text{si } x < t \end{cases}$$

En la función signo, el limite o valor umbral se encuentra en el punto cero. Si el valor procedente de la función base h es inferior a cero, el valor de salida es -1, en caso contrario la salida es 1:

$$signo(x) = \begin{cases} 1 & \text{si } x \geq 0 \\ \text{-1} & \text{si } x < 0 \end{cases}$$

Por último, la función sigmoidea se emplea cuando se desea una activación más suavizada y no una función escalonada como en los casos anteriores. El valor de salida viene dado por la siguiente expresión:

sigmoide(x) = $\frac{1}{1+e^{-x}}$

Las unidades neuronales también pueden funcionar como puertas lógicas [79], tal como se muestra en la figura 28.42. De esta forma, es posible construir redes que calculen cualquier función booleana de las entradas.

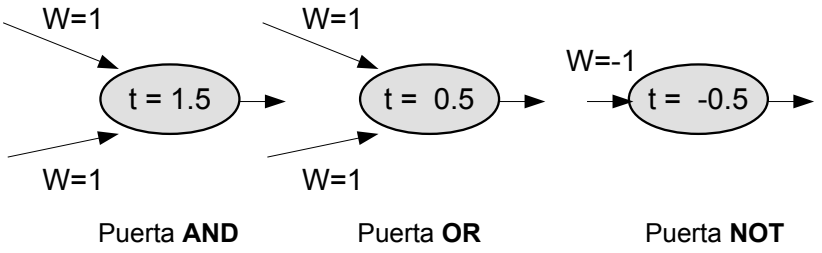


Figura 28.42: Unidades con función escalón que actúan como puertas lógicas.

Por último, cabe mencionar una de las principales características de las redes neuronales: su alto nivel de adptabilidad. Mediante procesos de aprendizaje es posible ajustar los pesos de las conexiones y adaptarse correctamente a cualquier situación. Dicho aprendizaje puede ser supervisado o automático. En el caso del aprendizaje supervisado se conocen las entradas de la red y como deberían ser las salidas para cada una de esas entradas. El ajuste de los pesos se realiza a partir del estudio de la salida real obtenida a partir de las entradas y de la salida esperada, intentando minimizar el error en la medida de los posible. Por otro lado, el aprendizaje automático dispone únicamente de los valores de entrada. Este tipo de algoritmos trata de ajustar los pesos hasta encontrar una estructura, configuración o patrón en los los datos de entrada.

Aplicaciones de las redes neuronales

Las redes neuronales se emplean principalmente para el aprendizaje de patrones y clasificación; en algunas ocasiones pueden ser una buena alternativa a sistemas complejos de reglas y máquinas de estados. A continuación se detalla una lista de aplicaciones generales y algunas de las aplicaciones en el desarrollo de videojuegos:

1. Aplicaciones generales
    - Reconocimiento facial
    - Análisis de imagen
    - Modelos financieros
    - Perfiles de mercado-cliente
    - Aplicaciones médicas (síntomas y diagnóstico de enfermedades)
    - Optimización de procesos industriales
    - Control de calidad
    - Detección de SPAM en correo electrónico
    - Reconocimiento de voz
    - Reconocimiento de símbolos en escritura
2. Aplicaciones en el desarrollo de videojuegos
    - Búsqueda de caminos

- Clasificación de personajes
- Sistemas de control y búsqueda de equilibrio: minimización de pérdidas o maximización de ganancias
- Toma de decisiones
- Elaboración de estrategias
- Simulaciones físicas

Estructuras de red

Existen multitud de tipos de estructuras de red y cada una de ellas otorga diferentes características de cómputo [79]. Todos ellos se pueden clasificar en dos grandes grupos: redes de **alimentación progresiva** y **redes recurrentes**. En el primer caso, las conexiones son unidireccionales y no se forman ciclos. Las neuronas de un nivel o capa sólo pueden estar conectadas con las del nivel siguiente. En la figura 28.43 se muestra un ejemplo de una red de alimentación progresiva con dos niveles; dicha red consta de dos entradas, dos neuronas en la capa oculta y una única salida. Al no existir ciclos en las conexiones, el cálculo se realiza de forma uniforme desde las unidades de entrada hasta las unidades de salida.

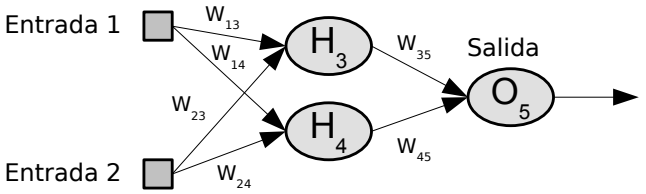


Figura 28.43: Red de alimentación progresiva de dos niveles: dos entradas, dos neuronas en la capa oculta y un nodo de salida

El cerebro no se puede considerar una red de alimentación progresiva porque si no, no tendría memoria a corto plazo; una red de recurrente, a diferencia de las de alimentación progresiva, permite mantener un estado interno. El problema de este tipo de redes es que el cálculo es más complejo, se pueden volver inestables y llegar a una conducta caótica.

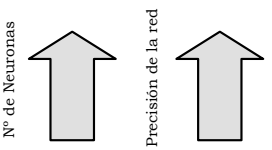


Figura 28.44: Relación entre número de neuronas y precisión de la red. La precisión suele aumentar a medida que se incrementa el número de neuronas.

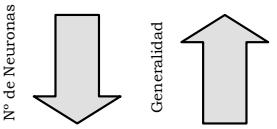


Figura 28.45: Relación entre número de neuronas y generalidad de la red. Cuanto menor es el número de neuronas más fácil es conseguir la generalidad.

Diseño de una red neuronal

A la hora de diseñar una red neuronal hay que resolver una serie de cuestiones importantes:

- Número de entradas
- Número de salidas
- Número de niveles
- Cantidad de neuronas en cada una de las capas o niveles intermedios.
- Cantidad de ejemplos de entrada para entrenar la red

A las dos primeras cuestiones es fácil encontrar una respuesta. Para la mayoría de problemas conocemos el número de entradas y las salidas deseadas. En el caso de encontrar ciertas dificultades a la hora de elegir el número de parámetros de entrada existe una relación directa con los patrones que pretendemos aprender en la red.

Parámetros = $log_2 P$

donde P es el número de patrones. Por otro lado, si queremos calcular un número de patrones estimado, se puede emplear la siguiente fórmula:

$$P = W \cdot 1/\varepsilon$$

donde W es el número de pesos en la capa intermedia y ε el error mínimo deseado en la red, dicho error se podría fijar en el siguiente intervalo: $0 \leq \varepsilon \leq 1/8$

Sin embargo, establecer una configuración adecuada en cuanto al número de niveles y neuronas es más complicado. Normalmente, la configuración se establece mediante un proceso de aprendizaje supervisado por prueba y error. En dicho proceso se conocen las entradas y las salidas que se deberían obtener ante tales entradas. Los errores producidos en la salida producen un ajuste de pesos para el correcto funcionamiento de la red. Si después del proceso de entrenamiento, no se obtiene el comportamiento deseado, es necesario modificar la configuración de la red.

Al diseñar una red neuronal siempre hay que tratar de encontrar un **equilibrio** entre *precisión* y *generalización*. Precisión se refiere a obtener el menor error posible en la salida ante los casos de entrada y generalización a la posibilidad de abarcar el mayor número de casos posibles. Cuando el número de neuronas en la capa oculta es alto, normalmente se consigue una mayor precisión a costa de un incremento de la complejidad (red más compleja implica mayor tiempo de entrenamiento).

Búsqueda de Equilibrio

En el diseño de una red neuronal hay que buscar siempre un equilibrio entre precisión y generalización

Por el contrario, cuando el número de neuronas es más bajo, obtenemos una red más simple que nos permite alcanzar una mayor generalización. El uso de funciones de activación sigmoidales también fomenta la generalización de la red. Además se debe tener especial cuidado a la hora de elegir el número de parámetros de entrada; cuanto mayor sea la cantidad de parámetros, mayores distinciones podremos hacer entre los individuos de una población, pero más complejo será alcanzar esa generalidad deseada.

Evidentemente, cuanto más complejo sea el problema a tratar, mayor será también el número de neuronas necesarias en la capa oculta. En casi la totalidad de los casos, la capa oculta necesita un menor número de neuronas que la capa de entrada. Según el teorema de Kolgomorov [58] "*El número de neuronas en la capa oculta debe ser como máximo dos veces el número de entradas*".

Para la configuración del número de neuronas en la capa oculta se pueden utilizar **algoritmos constructivos**. Básicamente, un algoritmo de este tipo parte de una configuración inicial en el que el número de neuronas es reducido. Se entrena la red hasta que el error se mantenga constante. Posteriormente, si no se ha alcanzado el error mínimo deseado, se puede agregar una neurona más a la capa intermedia con un peso pequeño y se repite el proceso anterior. Es importante tener en cuenta, que la agregación de neuronas no va a solucionar el problema si: i) el conjunto de datos de entrada es insuficiente, ii) hay datos que no se pueden aprender.

Para el cálculo del error de salida en función de las entradas se puede utilizar el error cuadrático medio:

$$e_{total} = \sqrt{e_1^2 + e_2^2 + ... + e_n^2}$$

donde e_i es el error obtenido en la salida para el ejemplo o patrón de entrada i. Para calcular el error cometido en uno de los patrones:

$1/n \sum_{j=0}(t_j - a_j)^2$

donde n es el número de neuronas, t el valor esperado en la salida y a_j el valor de salida real.

Finalmente, el proceso de entrenamiento y configuración de una red neuronal se debe detener cuando:

1. El error promedio alcanza el error mínimo deseado
2. El error no diminuye o lo hace de forma insignificante
3. El error comienza a incrementarse de forma gradual y constante.

Ajuste de los pesos de una red mediante aprendizaje *Hebbiano*

En esta sección se explica como se pueden ajustar los pesos de una red mediante la regla de aprendizaje de Hebb. Para ello supongamos una red sencilla constituida por dos entradas y una capa de salida formada por una única neurona. El valor umbral de activación de la neurona se va a ajustar de forma dinámica y para ello se considerará como una entrada más con peso negativo. El esquema de dicha red se muestra en la figura 28.46

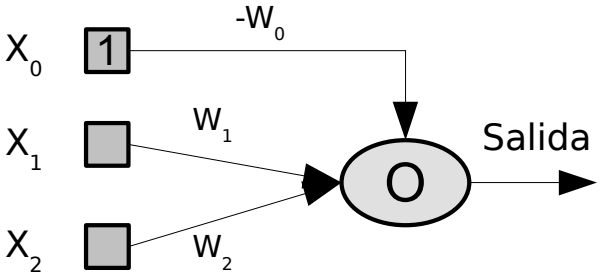


Figura 28.46: Esquema de la red empleada en el ejemplo de aprendizaje Hebbiano

La condición de activación de la neurona de salida vendrá dada por la siguiente expresión: $X_0W_0 + X_1W_1 + X_2W_2 > 0$. Ahora supongamos que deseamos modelar una red neuronal que represente una caja negra que simule el mismo comportamiento que una puerta lógica AND:

Casos	X_1	X_2	Salida
caso 1	0	0	0
caso 2	0	1	0
caso 3	1	0	0
caso 4	1	1	1

Cuadro 28.5: Comportamiento deseado en la red neuronal

En el año 1949, Donald Hebb observó que la sinapsis se reforzaba si la neurona de entrada y la de salida eran activadas de manera continua. Así, las conexiones en las que se produce este fenómeno son las que se refuerzan. Si trasladamos esta idea a una red neuronal artificial, las conexiones en las que la entrada y la salida se activan verán reforzados sus pesos, mientras que en el caso contrario se quedan tal cual. Por tanto, para entrenar la red se seguirán las siguientes reglas:

1. Si la salida es la esperada en función de las entradas, no se realizan ajustes en los pesos.
2. Si la salida es 1, pero debería ser 0, se reducen los pesos de las conexiones activas de acuerdo a un valor constante.
3. Si la salida es 0, pero debería ser 1, se aumentan los pesos de las conexiones activas de acuerdo a la misma constante.

Para entrenar la red se toman de forma aleatoria los casos expuestos anteriormente y se introducen en las entradas. Supongamos que inicialmente todos los pesos valen cero (tal como se comentó anteriormente, el valor inicial de los pesos debe ser bajo). Supongamos también, que el valor de incremento/decremento empleado para los pesos es 1.

Si el caso 1: $X_1 = 0, X_2 = 0$ fuera el primero en utilizarse para entrenar la red, $X_0W_0 + X_1W_1 + X_2W_2 = 0$, debido a que todos los pesos valen cero. Por tanto, la neurona de salida no se activa. En realidad es el valor esperado y, por tanto, no se producen modificación de los pesos. En realidad, el único ejemplo o caso que puede producir un error con todos los pesos a cero, es el caso 4.

Supongamos que el caso 4 es el siguiente ejemplo empleado para entrenar la red. $X_1 = 1, X_2 = 1$ y la salida esperada es 1; sin embargo $X_0W_0 + X_1W_1 + X_2W_2 = 0$. Se esperaba una activación de la salida que no se ha producido y, así, es necesario modificar los pesos incrementando los pesos de las neuronas activadas (en este caso todas las entradas):

- $W_0 = W_0 + C = 0 + 1 = 1$
- $W_1 = W_1 + C = 0 + 1 = 1$
- $W_2 = W_2 + C = 0 + 1 = 1$

De forma iterativa se van aplicando los casos de entrada hasta que no se producen errores en la salida. Finalmente, para este ejemplo, el ajuste apropiado de los pesos es el siguiente: $W_0 = -2$, $W_1 = 1$, $W_2 = 2$.

Aprendizaje supervisado mediante retropropagación (*backpropagation*)

Cuando tenemos más de un nivel en la red (ver figura 28.47), el método anterior resulta insuficiente y es necesario utilizar otros como el de *retropropagación*. Una de las condiciones indispensables para poder aplicar este método, es que las funciones de activación empleadas en las neuronas deben ser derivables. Básicamente se calcula el error producido en la salida y lo propaga hacia las capas intermedias para ajustar los pesos. El algoritmo consta de los siguientes pasos:

1. Definir una configuración inicial de la red.
2. Inicializar los pesos con valores aleatorios pequeños
3. REPETIR
   - Elegir un patrón y ubicarlo en la entrada de la red
   - Simular el funcionamiento de la red y observar la activación de neuronas en la capa oculta y la salida
   - Calcular el error producido en la salida

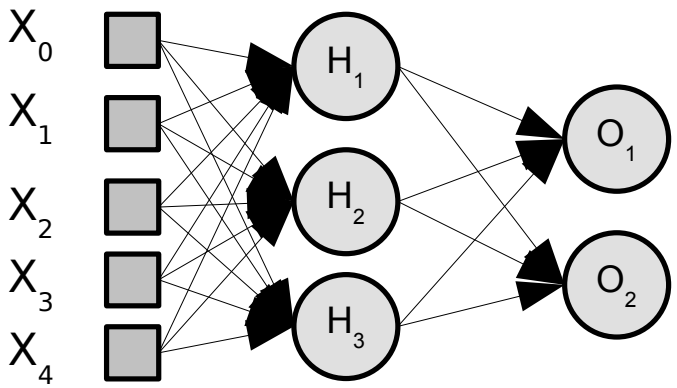


Figura 28.47: Esquema de un perceptrón o red neuronal multicapa

- Calcular las derivadas parciales del error con respecto a los pesos de salida (unión de la capa oculta y la de salida)
- Calcular las derivadas parciales del error con respecto a los pesos iniciales (unión de las entradas con la capa oculta)
- Ajustar los pesos de cada neurona para reducir el error.

4. FIN REPETIR
5. Continuar hasta minimizar el error.

En el caso de tener un solo nivel, la actualización de los pesos en función del error es sencilla:

Error = valor esperado (T) - valor real obtenido (O)

Si el error es positivo, el valor esperado es mayor que el real obtenido y por tanto será necesario aumentar O para reducir el error. En cambio, si el error es negativo habrá que realizar la acción contraria: reducir O.

Por otro lado, cada valor de entrada contribuirá en la entrada de la neurona según el peso: $W_j I_j$ (peso W y valor de entrada I). De esta forma, si I_j es positivo, un aumento del peso contribuirá a aumentar la salida y, si por el contrario, la entrada I_j es negativa, un aumento del peso asociado reducirá el valor de salida O. El efecto deseado se resume en la siguiente regla:

$$W_j = W_j + \alpha \times I_j \times Error$$

donde Error = $T - O$ y α es la *constante de velocidad de aprendizaje* (velocidad de convergencia). Normalmente esta constante oscila entre 0.05 y 0.25. El valor de α se debe disminuir a medida que se reduce el error.

En el caso de tener varios niveles, la actualización de pesos es un poco más compleja (ver figura 28.48). Los pesos se actualizan a partir del error cometido, dicho error se propaga desde las primeras capas hasta las finales. La idea general es evaluar las consecuencias del error y dividirlas entre todos aquellos que hayan contribuido a dicho error. En primer lugar se actualizan los pesos de las conexiones entre las neuronas de la capa oculta y la de salida. Las salidas a_j de las neuronas de la capa oculta supondrán la entrada de las neuronas de salida.

Figura 28.48: Red de alimentación progresiva de dos niveles

$$W_{j,i} = W_{j,i} + \alpha \times a_j \times \Delta_i$$

en donde Δ_i es igual a $Error_i \times g'(entrada_i)$, y g' es la derivada de la función de activación g.

Una vez actualizadas las conexiones entre la capa oculta y la de salida, queda por actualizar las conexiones entre las entradas y la capa oculta. Para la actualización se necesita un valor de error y es aquí donde tiene lugar la propagación posterior. La idea es que el nodo oculto j es responsable de una parte del error producido en Δ_i en cada uno de los nodos de salida con los que conecta. Es decir, todos los nodos de la capa oculta tienen parte de culpa en el error producido en las salidas; de esta forma, los valores de Δ_i son divididos de acuerdo a la intensidad de conexión entre los nodos ocultos y las salida, y se propaga hacia atrás para calcular el valor Δ_j del nodo oculto. Δ_j se calcula de la siguiente forma:

$$\Delta_j = g'(entrada_j) \sum_i W_{j,i} \Delta_i$$

La regla de actualización de pesos quedaría definida de la siguiente manera:

$$W_{k,j} = W_{k,j} + \alpha \times I_k \times \Delta_j$$

Ejercicio propuesto: Dada una red neuronal con una única neurona oculta, con patrones de entrada P1 (1 0 1), P2 (1 1 0), P3 (1 0 0), llevar acabo un proceso de entrenamiento mediante *backpropagation* para ajustar los pesos de la red y que ésta se comporte como la función XOR

Aplicación de redes neuronales al juego de los asteroides

Las redes neuronales se han aplicado con frecuencia en el juego de los asteroides, en el que una nave debe evitar el impacto con asteroides que se mueven de forma aleatoria es un espacio común. La red neuronal permite modelar el comportamiento de la nave para evitar el impacto con los asteroides más cercanos. En [83] se plantea una configuración de red neuronal para solucionar el problema. Dicha configuración consta de cuatro entradas, una capa oculta con 8 neuronas y una capa de salida de tres neuronas, donde cada una representa una posible acción.

Las dos primeras entradas corresponden con la componente x e y del vector entre la nave y el asteroide más cercano. La tercera entrada corresponde a la velocidad con la que la nave y el asteroide más cercano se aproximan. Finalmente, la cuarta entrada representa la dirección que está siguiendo la nave.

La salida de la red neuronal son tres posibles acciones: i) aceleración puntual (turbo), ii) giro a la izquierda, iii) giro a la derecha. En [83] se encuentra publicada una posible implementación al problema.

Ejercicio propuesto: Diseñar una red neuronal que regule el comportamiento de la nave en el problema planteado, y evite la colisión con los asteroides

Figura 28.49: Juego de asteroides. La nave debe evitar el impacto con los asteroides en movimiento, con diferentes formas, tamaños y velocidades.

28.3. Algoritmos de búsqueda

28.3.1. Problemas y soluciones.

Como ya se ha visto en la sección 28.1.4, una de las formas más habituales de representar el comportamiento de un agente 28.1.3 es mediante un sistema de estados y transiciones. Este formalismo permite la descripción perfecta de cualquier actividad que tenga que desplegar un determinado agente en todo momento del videojuego.

La actividad desarrollada por cada agente, viene dada por la **secuencia** de transiciones o **acciones** realizadas hasta dicho momento. Pongamos por ejemplo un problema lógico clásico como es siguiente:

Sea una par de islas llamadas **IslaA** e **IslaB**. En IslaA hay una **oveja**, una **col**, un **lobo**, una barca y un **barquero**. La cuestión a resolver es cómo llevar al lobo, la oveja y la col a IslaB sin perder a nadie en el intento. Si eres el barquero has de considerar que: sólo puedes trasladar un objeto o animal en cada momento en la barca, que la oveja no se puede quedar sola con la col, ya que se la comería, y que igual destino correría la oveja a solas con el lobo.

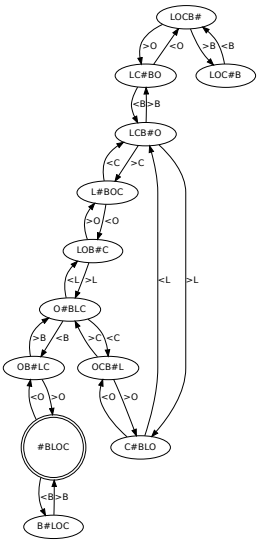


Figura 28.50: Espacio de estados del problema de la isla

En un primer momento para trabajar el problema propuesto definimos los elementos que van a configurar un estado, así como las acciones que podríamos hacer en cada uno de ellos. La forma de describir una foto instantánea de una situación es mediante la enumeración de los elementos que hay en cada isla. Llamemos al lobo **L**, a la oveja **O**, a la col **C** y al barquero junto con la barca **B**, y determinemos que IslaA está a la izquierda y IslaB está a la derecha. De esta manera representarmos cualquier situación del problema mediante una cadena de caracteres del tipo **LOCB#** que refleja la situación inicial donde el lobo, oveja, la col y el barquero están en IslaA, o por ejemplo esta otra situación **LC#OB** donde el barquero y la oveja se encuentran en IslaB y el lobo y la col en IslaA.

Una vez fijados los estados de nuestro problema de la isla, tendremos que enumerar las acciones que se pueden desarrollar en un determinado estado. Pongamos por ejemplo el estado **LOCB#**, en esta situación las acciones que podemos realizar son: coger la barca e ir de vacio a IslaB, que representamos mediante la secuencia de caracteres **>B**, llevar el lobo a IslaB **>L**, llevar la oveja **>O** o llevar la col **>C**. Considerando solamente las acciones válidas, que no provocan la pérdida de algún elemento, tendremos que en el estado **LOCB#** sólo son correctas las acciones: **>B** y **>O**.

Para definir este conjunto de acciones establemos la función **sucesor** de un estado como el conjunto de pares **(acción,estado)** como los estados que son accesibles desde el mismo mediante una acciónn concreta. Por ejemplo **sucesor**(LOCB#) = ((>B,LOC#B),(>O,LC#BC)). En estos momentos hemos definido el espacio de estados del problema (28.50).

El espacio de estados de un problema queda definido mediante el conjunto de estados posibles del problema y la función sucesor de cada uno. Este espacio de estado se puede representar mediante un grafo donde los nodos son los estados y los arcos son las acciones.

Establecido un espacio de estados, un **problema** [79] es llegar de un **estado inicial** a un **estado meta** deseado mediante una secuencia de acciones válidas. En el ejemplo de la isla será búscar la secuencia de acciones que llevase del estado inicial LOCB# al final #BLOC.

Los componentes de un **problema** son:

1. Un espacio de estados.
2. Un estado inicial.
3. Una función meta, que identifique estados finales.

El fichero problema.py contiene, en lenguaje python, una realización de estos conceptos mediante la especificación de una clase abstracta **Problema**, que representa cualquier problema en terminos de espacio de estados, y una clase derivada **ProblemaIsla** para el problema de la isla.

Listado 28.8: class Problema

```
class Problema:
    def __init__(self,estadoInit,nombre="NoDef"):
        self.id=nombre
        self.estInit=estadoInit

    def estadoValido(self,estado):
        pass

    def sucesor(self,estado):
        pass

    def meta(self,estado):
        pass

    def estadoInit(self):
        return self.estInit
```

En los términos fijados, una **solución** a un problema es, precisamente, un camino o secuencia de acciones válidas que dirige al sistema de un estado inicial a un estado meta. Al proceso de buscar entre todos los caminos posibles, que parten de un estado inicial establecido, aquellos que son soluciones se denomina **resolución de problemas mediante técnicas de búsqueda**. Una solución para el problema de la isla se puede ver en la figura 28.51.

Listado 28.9: class ProblemaIsla

```
class ProblemaIsla(Problema):
"""Problema de la Isla.
    Un estado es un string del tipo LOCB#.
    L representa LOBO
    O representa OVEJA
    C representa COL
    B representa al Barquero y la Barca.
    La izquierda de # es lo que hay en la isla A.
    La derecha de # es lo que hay en la isla B."""

    def __init__(self,estado):
        Problema.__init__(self,estado,"Isla")
        self.g={}
        self.g["LOCB#"]=[(">O","LC#BO"),(">B","LOC#B")]
        self.g["LOC#B"]=[("<B","LOCB#")]
        self.g["LC#BO"]=[("<O","LOCB#"),(">B","LCB#O")]
        self.g["LCB#O"]=[("<C","L#BOC"),(">L","C#BLO")]
        self.g["L#BOC"]=[("<C","LCB#O"),("<O","LOB#C")]
        self.g["LOB#C"]=[(">L","O#BLC"),(">O","L#BOC")]
        self.g["O#BLC"]=[("<L","LOB#C"),("<B","OB#LC"),("<C","OCB#L
            ")]
        self.g["OCB#L"]=[(">C","O#BLC"),(">O","C#BLO")]
        self.g["OB#LC"]=[(">B","O#BLC"),(">O","#BLOC")]
        self.g["C#BLO"]=[("<L","LCB#O"),("<O","OCB#L")]
        self.g["#BLOC"]=[("<O","OB#LC"),("<B","B#LOC")]
        self.g["B#LOC"]=[(">B","#BLOC")]

    def estadoValido(self,estado):
        return estado in self.g.keys()

    def sucesor(self,estado):
        if self.estadoValido(estado):
            return self.g[estado]
```

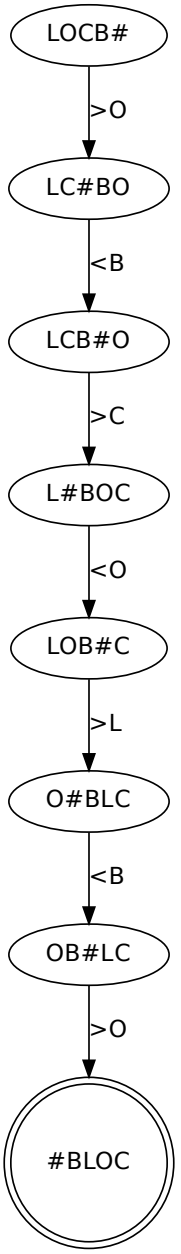


Figura 28.51: Una solución al problema de la isla.

```
33          else:
34              return None
35
36      def meta(self,estado):
37          return estado=="#BLOC"
```

28.3.2. Organizar la búsqueda de soluciones

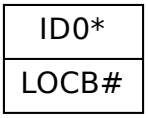


Figura 28.52: Árbol inicial.

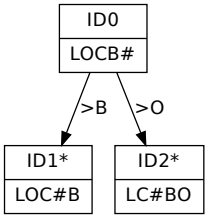


Figura 28.53: Primera expansión.

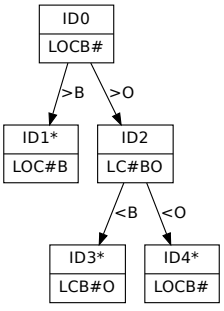


Figura 28.54: Segunda expansión.

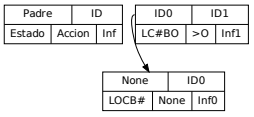


Figura 28.55: Nodo del árbol de búsqueda

La solución a un problema es un camino por el espacio de estados, que permite transformar la realidad de una situación inicial a una situación deseada o meta. El proceso de búsqueda deberá explorar el conjunto de caminos posibles y centrarse en aquellos que son solución. Esto marca un primer objetivo que es la obtención de forma organizada y sistemática de todos los posible caminos que pudieran existir en un espacio de estados.

Una estructura de datos, que permite realizar este propósito es el árbol. Un árbol puede incorporar la información relativa a un estado en sus nodos, así como las acciones realizadas, y etiquetar los enlaces entre un padre y sus hijos. Un árbol cuyo nodo raíz represente el estado inicial de un problema y la relación filial entre nodos hijos y padres venga explicitada por el conjunto de acciones posibles según el espacio de estados se denomina **árbol de búsqueda**, y cada una de sus ramas representa un camino posible.

El árbol de búsqueda es una estructura que permite generar y almacenar todos los posibles caminos de un espacio de estado.

Un proceso iterativo de generación del árbol de búsqueda es el siguiente algoritmo.

Listado 28.10: Crear árbol de búsqueda

```
1 while exista un nodo hoja sin explorar:
2     tomar n como nodo a explorar de los nodos hoja no explorados
3     if n representa un estado meta:
4         solucion es la rama desde la raiz al nodo n
5         salir.
6     else:
7         explorar el n
8         marcar n como explorado.
```

Siguiendo este algoritmo, como se muestra en la figura 28.52, el problema de la isla comenzaría con un árbol con un único nodo raíz y hoja. Observando el espacio de estados de la figura 28.50 se explora el nodo ID0 obteniendo el árbol de la figura 28.53. En la tercera iteración y seleccionando, por ejemplo, el nodo ID2 tendremos el árbol de búsqueda que muestra la figura 28.54.

Antes de continuar desarrollando el esbozo de algoritmo expuesto, definiremos los elementos que deberán constituir un nodo del árbol de búsqueda como se muestra en la figura 28.55. Estos son:

1. **ID** Un identificador único para el nodo.
2. **Padre** Una referencia u enlace al nodo padre.

3. **Estado** La representación del estado.
4. **Acción** La acción realizada para generarse.
5. **Inf** Información diversa referente a la situación de la búsqueda, valoraciones, etc.

Una posible realización del concepto de nodo del árbol de búsqueda se muestra en la clase que se lista a continuación. En esta clase **class NodoArbol** se crean un nodo con los atributos que hemos indicado. También se facilitan los métodos para la construcción del camino desde el nodo raiz del árbol a un nodo concreto (línea 19), y la creación de un hijo (línea 28). Se le ha añadido una valoración general del nodo **val**, que se usará posteriormente.

Listado 28.11: Clase para el nodo del Árbol de búsqueda

```
1  class NodoArbol:
2      id=0
3      def __init__(self,padre,acc,estado,inf=None,val=0):
4          self.padre=padre
5          self.accion=acc
6          self.estado=estado
7          self.inf=inf
8          self.id=NodoArbol.id
9          self.val=val
10         NodoArbol.id=NodoArbol.id+1
11
12     def __str__(self):
13         if self.padre==None:
14             idpadre=-1
15         else:
16             idpadre=self.padre.id
17         return "[%i|%i,%s,%s,%s]"%(idpadre,self.id,self.accion,self
               .estado,self.inf)
18
19     def camino(self):
20         """Retorna la lista de nodos hasta el nodo raiz."""
21         p=[]
22         n=self
23         while n<>None:
24             p.insert(0,n)
25             n=n.padre
26         return p
27
28     def GeneraNodoHijo(self,ac,est):
29             n=NodoArbol(self,ac,est,{'Prof':n.inf['Prof']+1})
30             n.val=n.inf['Prof']
31             return n
```

El algoritmo descrito selecciona, del conjunto de nodos hoja del árbol de búsqueda, un nodo para analizar. Primero verifica que no es una solución y posteriormente lo analiza. Este análisis consiste en obtener a partir del problema, con la función sucesor, el conjunto de nuevos nodos hijos e introducirlos en el árbol de búsqueda como nuevas hojas al mismo tiempo que el nodo estudiado deja de ser hoja.

De lo descrito anteriormente el proceso de selección del nodo hoja a **expandir** es de una importancia , ya que de una buena o mala selección dependerá que la búsqueda de un nodo meta sea más o menos eficiente.

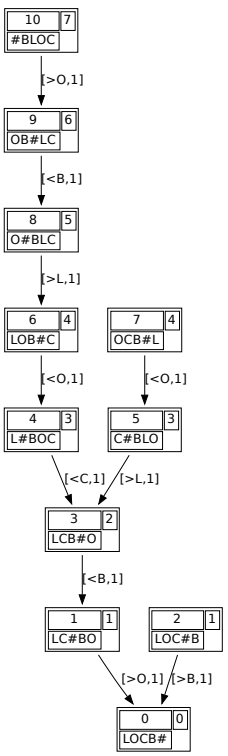


Figura 28.56: Árbol de búsqueda completo Anchura.

Al mecanismo seleccionado para la elección del nodo hoja que hay que expandir se denomina **estrategia de búsqueda**.

Si no se tiene más información, para un problema, que el espacio de estado, las estrategias de búsqueda están limitadas a dos:

1. **Anchura** Las hojas son seleccionadas por antiguedad, es decir, los nodos de menor profundidad.
2. **Profundida** Las hojas son seleccionadas por novedad, es decir, los nodos de mayor profundidad.

Una implentación de las estructuras necesarias para realizar estas dos estrategias son mostradas en las siguientes clases: **AlmacenNodos**, **PilaNodos** y **ColaNodos**.

Listado 28.12: Clases de Cola Pila

```
1  class AlmacenNodos:
2      """Modela la frontera."""
3      def __init__(self,nombre="NoName"):
4          self.id=nombre
5      def add(self,elem):
6          """Introduce un elemento."""
7          pass
8      def esVacia(self):
9          """Retorna True si no tiene ningun elemento."""
10         pass
11     def get(self):
12         """Retorna un elemento."""
13         pass
14     def esta(self,id):
15         """Retorna True si el elemento con Id esta en la estructura
               ."""
16     pass
17
18 class PilaNodo(AlmacenNodos):
19     """Modela un Almacen como una Pila: Ultimo en entrar, primero
           en salir."""
20     def __init__(self,nombre):
21         AlmacenNodos.__init__(self,nombre)
22         self.p=[]
23         self.lest=[]
24     def esVacia(self):
25         return len(self.p)==0
26     def add(self,elem):
27         self.p.append(elem)
28         self.lest.append(elem.estado)
29     def get(self):
30         return self.p.pop()
31     def esta(self,estado):
32         return estado in self.lest
33
34 class ColaNodo(AlmacenNodos):
35     """ Modela un Almacen como una Cola: Primero en entrar, primero
             en salir."""
36     def __init__(self,nombre):
37         AlmacenNodos.__init__(self,nombre)
38         self.c=[]
39         self.lest=[]
40     def esVacia(self):
41         return len(self.c)==0
```

```
42      def add(self,elem):
43          self.c.insert(0,elem)
44          self.lest.append(elem.estado)
45      def get(self):
46          return self.c.pop()
47      def esta(self,estado):
48          return estado in self.lest
```

En el siguiente código se muestra el algoritmo de búsqueda, y como dependiendo del parámetro introducido produce la salida con una estrategia de anchura o de profundidad. Aplicado al problema de la Isla los resultados obtenidos según las estrategias utilizadas son:

- Anchura. (None,LOCB#), (>0,LC#BO), (<B,LCB#O), (<C,L#BOC), (<O,LOB#C), (>L,O#BLC), (<B,OB#LC), (>O,#BLOC). (figura 28.56).
- Profundidad. (None,LOCB#), (>O,LC#BO), (<B,LCB#O), (>L,C#BLO), (<O,OCB#L), (>C,O#BLC), (<B,OB#LC), (>O,#BLOC). (figura 28.57).

Listado 28.13: Algoritmo de búsqueda

```
1  def BuscaSolucionesSinInf(prob,estrategia):
2      if estrategia=='PROFUNDIDAD':
3          frontera=PilaNodo('Profundidad')
4      elif estrategia=='ANCHURA':
5          frontera=ColaNodo('Anchura')
6      n=NodoArbol(None,None,prob.estadoInit(),{'Prof':0})
7      solucion=None
8      frontera.add(n)
9      while (not frontera.esVacia()) and (solucion==None):
10         n=frontera.get()
11         if prob.meta(n.estado):
12             solucion=n
13         else:
14             for ac,est in prob.sucesor(n.estado):
15                 if not frontera.esta(est):
16                     h=GeneraNodo(n,ac,est)
17                     frontera.add(h)
18     return solucion
```

Sobre la eficiencia de estas estrategias hay que destacar que la estrategia de búsqueda en anchura es una estrategia completa, esto es que es capaz de encontrar la solución si existe; pero en el peor de los casos se ha de sarrollar el árbol completo con una complejidad temporal (tiempo de ejecución) del orden $O(b^n)$ para un problema con b acciones en cada estado y una logitud de n acciones hasta un estado meta. La estrategia de profundidad tiene el problema que se puede perder si el árbol de búsqueda es infinito, y no podría garantizar encontrar solución alguna; pero si la encuentra en el camino seleccionado la necesidad de almacenamiento es la menor posible $O(n)$ que es la longitud de la solución.

Para evitar los problemas de perderse por las profundidades de los árboles de búsqueda en estrategias de profundidad, existe la opción de incorporar un **límite de profundidad** y definir una estrategia **profundidad acotada**. Si en una determina cota de la estrategia de profundidad acotada y habiendo expandido el árbol completo hasta dicha cota no se encontrase ninguna solución, por que ésta se encontrase por debajo de dicha cota, se podría volver a comenzar aumentado dicha cota en alguna cantidad. La estrategia que modifica la cota límite de una estrategia en profundidad se denomina **estrategia de profundidad iterativa**.

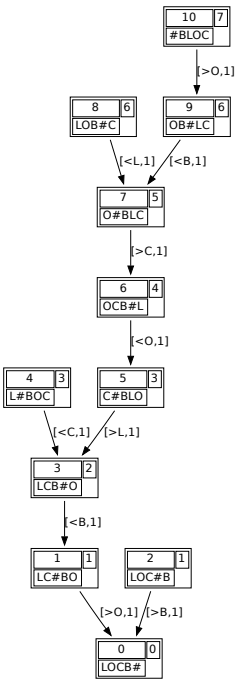


Figura 28.57: Árbol de búsqueda completo Profundidad.

28.3.3. Búsqueda con información

Las estrategias de anchura, profundidad y sus derivadas encuentran una solución al problema planteado. Esta solución puede que no sea única, esto es, que existan una multitud de posibles soluciones; pero estas estrategias obtiene como solución aquella que primero encuentran. En este tipo de problemas sólo son elevantes las soluciones y no su configuración, cualquier solución será válidad siempre que lleve a una situación meta.

La situación cambia si tuviéramos que abonar una cantidad de dinero por cada acción que realizamos, en este caso está claro que entre las soluciones obtenidas para un problema optaríamos por la solución en anchura frente a la de profundidad. Esta decisión se toma en base al conocimiento que una solución obtenida mediante una estrategia en anchura será la de menor profundidad encotrada, esto es, la que menos pasos tendrán que realizar.

Esta situación donde cada acción cuesta igual no es la más normal, por contra en la gran mayoría de las situaciones reales cada una de las acciones que podamos realizar en un estado concreto dependerá del tipo de acción, así como de otras características asociadas al problema concreto. En definitiva, el coste de cada acción normalmente es difente.

En situaciones con diferentes costos, una estrategia de anchura no garatiza que la solución obtenida sea la **mejor** con un coste menor. Para incorporar estos criterios al algoritmo de búsqueda, se definirá una valoración entre las distintas soluciones que permita seleccionar aquella que la maximize.

Una de las valoraciones más sencillas a considerar para una solución es la suma de cada uno de los costes de las acciones que la configuran. Para poder establecer esta valoración añadimos en cada acción posible en un estado el coste asociado a su realización como un valor numérico positivo. La nueva función sucesor devolverá una tupla (Acc,val,estado) y definiremos el costo de un nodo n' obtenido mediante la aplicación de la acción Acc a un nodo n como una función no decreciente $C(n)$:

$$(\{Acc, val\}, n') \in sucesor(n) C(n') = C(n) + val \tag{28.3}$$

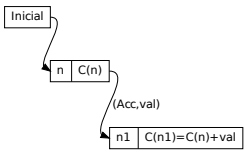


Figura 28.58: Costo asociado a un nodo.

Ahora ya tenemos un valor para decidir que nodo hoja coger, el menor valor de costo de los nodos hoja, o que el conjunto de nodos hojas estén ordenados de forma creciente por el valor del costo.

Con el objeto de generalizar esta estrategia y poder utilizar las estructuras posteriormente, se ha definido una nueva estructura de almacen como una lista ordenada por una valor del nodo. La estructura elegida para realizar la cola ordenada ha sido el montículo, una estructura vectorial donde la posición inicial del vector se encuentran siempre el elemento menor.

Listado 28.14: Montículo como lista ordenada

```
1  import heapq
2  class ListaOrdenada(Almacen):
3      def __init__(self,nombre):
4      Almacen.__init__(self,nombre)
5      self.p=[]
6
7      def esVacia(self):
8          return len(self.p)==0
9
10     def add(self,elem):
11         print elem
12     valor,n=elem
13     heapq.heappush(self.p,(valor,n))
```

```
14
15      def get(self):
16          return heapq.pop(self.p)
```

A la estrategia que selecciona un nodo hoja con valor de costo menor se denomina **estrategia de costo uniforme.**

Como ejemplo definimos un nuevo problema. El problema es el conocido como 8 puzzle donde se tiene un artilugio con 8 piezas numeradas del 1 al 8 y un espacio sin pieza. Las piezas se puden deslizar al espacio en blanco y así cambiar la configuración del puzzle. Si se representa una configuración concreta del 8 puzzle con una cadena de 9 caracteres donde los 3 primeros representan la primera fila, y así sucesivamente y el espacio en blanco por el caracter B, y consideramos que los movimientos son los del espacio blanco. Si consideramos que de los cuatro movimientos posibles el costo de los movimientos hacia arriba e izquierda van a costar 2 unidades frente a la unidad de los movimientos abajo y derecha.

El problema propuesto es definido en el siguiente código ejemplo.

Listado 28.15: Problema del 8 puzzle.

```
1  class Problema8Puzzle(Problema):
2
3      def __init__(self,estadoI,estadoFin,h='DESCOLOCADAS'):
4          Problema.__init__(self,estadoI,"8Puzzle")
5          self.estadoFin=estadoFin
6          self.h=h
7
8      def estadoValido(self,estado):
9          valido=True
10         for ficha in range(1,9):
11             valido=valido and (str(ficha) in estado)
12         valido=valido and (len(estado)==9)
13         valido='B' in estado
14         return valido
15
16     def sucesor(self,estado):
17         if self.estadoValido(estado):
18             PB=estado.find('B')
19             FB=PB/3
20             CB=PB%3
21             mv=[]
22             if FB<2:
23                 NP=(FB+1)*3+CB
24                 est=list(estado[:])
25                 est[PB]=est[NP]
26                 est[NP]='B'
27                 mv.append(({'Acc':'AB','val':1},"".join(est)))
28             if FB>0:
29                 NP=(FB-1)*3+CB
30                 est=list(estado[:])
31                 est[PB]=est[NP]
32                 est[NP]='B'
33                 mv.append(({'Acc':'AR','val':2},"".join(est)))
34             if CB<2:
35                 NP=FB*3+CB+1
36                 est=list(estado[:])
37                 est[PB]=est[NP]
38                 est[NP]='B'
39                 mv.append(({'Acc':'DE','val':1},"".join(est)))
40             if CB>0:
41                 NP=FB*3+CB-1
42                 est=list(estado[:])
43                 est[PB]=est[NP]
44                 est[NP]='B'
45                 mv.append(({'Acc':'IZ','val':2},"".join(est)))
46             return mv
```

```
47          else:
48              return None
49
50      def meta(self,estado):
51          return estado==self.estadoFin
```

Para un problema del 8puzzle donde la configuración inicial consiste en **1342B5786** y la final **12345678B** el algoritmo de búsqueda con la estrategia de costo uniforme devuelve la solución siguiente:

1342B5786

1. ({'Acc': 'IZ', 'val': 2},134B25786)
2. ({'Acc': 'AB', 'val': 1},134725B86)
3. ({'Acc': 'DE', 'val': 1},1347258B6)
4. ({'Acc': 'DE', 'val': 1},13472586B)
5. ({'Acc': 'AR', 'val': 2},13472B865)
6. ({'Acc': 'AR', 'val': 2},13B724865)
7. ({'Acc': 'IZ', 'val': 2},1B3724865)
8. ({'Acc': 'AB', 'val': 1},1237B4865)
9. ({'Acc': 'DE', 'val': 1},12374B865)
10. ({'Acc': 'AB', 'val': 1},12374586B)
11. ({'Acc': 'IZ', 'val': 2},1237458B6)
12. ({'Acc': 'IZ', 'val': 2},123745B86)
13. ({'Acc': 'AR', 'val': 2},123B45786)
14. ({'Acc': 'DE', 'val': 1},1234B5786)
15. ({'Acc': 'DE', 'val': 1},12345B786)
16. ({'Acc': 'AB', 'val': 1},12345678B)

12345678B

El coste de esta solución es de 23 que podremos asegurar que es el menor coste posible, o en otras palabras, que **no existe otra solución con un coste menor**.

AL considerar el coste que se sufre al generar un nodo, hemos asegurado que el resultado que obtenemos es el mejor de todos; pero no se ha solucionado el problema de la complejidad. La complejidad de la estrategia de costo uniforme se puede equiparar a la de estrategia en anchura haciendo que cada paso sea un incremento Ic del costo $O(b^{(n/Ic)})$.

Considerando que la mejor estrategia, siempre que no se pierda, es la estrategia en profundidad una opción muy interesante podría ser combinar la estrategia de costo uniforme con algo parecida a la estrategia en profundidad. Para poder llevar acabo esta estrategia deberíamos ser capaces de decidirnos por la acción que mejor nos lleve a una solución. En otras palabras, tendríamos que ser capaces de poder **evaluar el coste que nos queda por realizar** desde la situación actual a una meta.

Este valor del costo que nos queda por realizar desde un estado para alcanzar un estado meta se denomina valor de la **heurística**.

Para añadir esta heurística modificaremos la definición de un problema incorporando una nueva función que evalue el valor de la heurística. Dado un estado n la función heuristica $h(n)$ será el valor estimado del costo que queda por realizar hasta un estado meta.

Listado 28.16: Problema Heurística

```
class Problema:
    def __init__(self,estadoInit,nombre="NoDef"):
        self.id=nombre
        self.estInit=estadoInit

    def estadoValido(self,estado):
        pass

    def sucesor(self,estado):
        pass

    def meta(self,estado):
        pass

    def estadoInit(self):
        return self.estInit

    def heuristica(self,estado):
        pass
```

En el problema del 8puzzle podemos estimar el número de movimientos que tendremos que hacer desde un estado hasta el estado meta con una cota mínima que sería el número de casillas descolocadas. Este valor nos impone una cota mínima para el coste que tendremos que hacer, ya que en una situación idílica con un único movimiento colocásemos cada ficha descolocada. Pero como tenemos que mover cada pieza una posición y sólo son válidos los movimientos horizontales y verticales, otra estimación un poco más exacta sería la suma de las distancias manhattan de cada pieza descolocada a su posición destino.

Incorporado los nuevos métodos a la clase del problema del 8 puzzle, el código queda como se muestra en el siguiente listado.

Listado 28.17: Problema 8Puzzle Heurística

```
class Problema8Puzzle(Problema):

    def __init__(self,estadoI,estadoFin,h='DESCOLOCADAS'):
    Problema.__init__(self,estadoI,"8Puzzle")
        self.estadoFin=estadoFin
        self.h=h

    def estadoValido(self,estado):
    valido=True
        for ficha in range(1,9):
        valido=valido and (str(ficha) in estado)
    valido=valido and (len(estado)==9)
    valido='B' in estado
        return valido

    def sucesor(self,estado):
    ................

    def meta(self,estado):
        return estado==self.estadoFin
    def posDistintas(self,estado1,estado2):
```

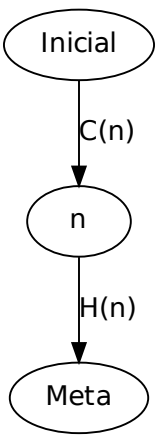


Figura 28.59: Información de un problema: Costo y Heurística

```
22      n=0
23          for i in range(len(estado1)):
24              if estado1[i]<>estado2[i]:
25          n=n+1
26          return n
27      def movDistintas(self,estado1,estado2):
28      n=0
29          for i in range(len(estado1)):
30              if estado1[i]<>estado2[i]:
31          py1=i/3
32                  px1=i%3
33                  pc=estado2.find(estado1[i])
34          py2=pc/3
35                  px2=pc%3
36                  n=abs(px2-px1)+(py2-py1)
37          return n
38
39      def setHeuristica(self,h='DESCOLOCADAS'):
40          self.h=h
41
42      def heuristica(self,estado):
43          if self.h=='DESCOLOCADAS':
44              return self.posDistintas(estado,self.estadoFin)
45          elif self.h=='CUNIFORME':
46              return 0
47          else:
48              return self.movDistintas(estado,self.estadoFin)
```

Llegados a este punto, y para combinar la información del costo desarrollado con la del costo por desarrollar o función heurística, definiremos el valor del nodo como la suma del costo más el valor de la función heurística.

$$f(n) = C(n) + H(n) \tag{28.4}$$

A la estrategia que ordena el conjunto de nodos hojas en función del valor f(n) 28.4 se denomina estrategia de búsqueda **A***. Formalmente para poder nombra esta estrategia como A* tendríamos que haber definido la función heurística como h^* la que **mejor** que estima el costo por gastar; pero como normalmente nuestra función h no es tan buena y minora a h^* tendríamos que denominar a esta estrategia como estrategia **A**.

Las características mas relevantes de esta estrategia A* son:

1. Es completa. Si existe solución la encuentra.
2. Es la mejor. La solución que encuentra es la de menor coste.
3. El número de nodos generados para encontrar la solución es el menor que en las estrategias anteriores.

Listado 28.18: Busca soluciones con informacion Heurística

```
1  def GeneraNodo(p,n,ac,est,estr='A'):
2      MUCHO=10000000
3      prof=n.inf['Prof']+1
4      coste=n.inf['Coste']+ac['val']
5      h=p.heuristica(est)
6
7      if estr=='A':       val=coste+h
8      elif estr=='PROF':  val=MUCHO-prof
9      elif estr=='ANCH':  val=prof
10     elif estr=='COST':  val=coste
11     else:               val=h
```

```
12      return NodoArbol(n,ac,est,{'Prof':prof,'Coste':coste,'h':h},val
            )
13
14  def BuscaSolucionesA(prob,estr='A'):
15      frontera=AlmacenOrdenado()
16      h=prob.heuristica(prob.estadoInit())
17      n=NodoArbol(None,None,prob.estadoInit(),{'Prof':0,'Coste':0,'h'
            :h},0)
18      solucion=None
19      frontera.add((n.val,n.estado,n))
20      while (not frontera.esVacia()) and (solucion==None):
21          n=frontera.get()
22          if prob.meta(n.estado):
23              solucion=n
24          else:
25              for ac,est in prob.sucesor(n.estado):
26                  if not frontera.esta(est):
27                      h=GeneraNodo(prob,n,ac,est,estr)
28                      frontera.add((h.val,h.estado,h))
29      return solucion
```

Las diferentes formas de actuar de las distintas estrategias se han plasmado en el problema del 8 puzzle estudiado con los valores que se muestran en la siguiente tabla.

Estrategia	Num de Nodos	S. Óptima
Profundidad	>30000	No
Anchura	16018	No
Costo Uniforme	14873	Si
A*	2326	Si

28.4. Planificación de caminos

En la actualidad, es muy difícil encontrar un videojuego que no contenga algún tipo de agente o personaje autónomo. Coches que circulan por carreteras, pájaros que vuelan, ratoncillos que corren para esconderse o malvados enemigos que nos persiguen. Estos personajes pueden hacer multitud de cosas; pero si hay una única acción que como mínimo se les puede exigir es la de poder dirigirse de un lugar a otro del mundo que habitan.

Este proceso de trasladarse de un lugar a otro se divide en un par de procesos más sencillos. El primero de ellos identifica el camino o lugares por donde el actor a de pasar, y el segundo intenta proceder trasladando al personaje siguiendo el plan trazado. Al primero se reconoce como el proceso de **planificación de recorridos** o caminos (pathfinder) y será el objeto de esta sección.

28.4.1. Puntos visibles

Sea un mundo definido en un videojuego de dos dimensiones, como el de la figura 28.60, donde habita un actor que se mueve entre los bloques. Para cada mapa, de este tipo, se puede crear un grafo como el que se muestra en la figura 28.62 donde los nodos son los puntos visibles para dicho mapa y los arcos son la línea de visión directa entre los puntos. El conjunto de puntos que nos permiten una visión completa del escenario o mapa son los denominados **puntos visibles** 28.61 que serán identificados por sus coordenadas (x,y) o por un identificador.

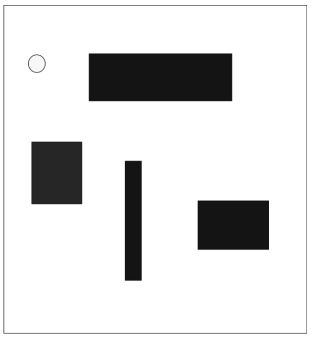


Figura 28.60: Escenario o mapa 2D.

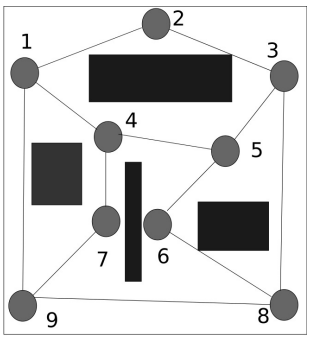


Figura 28.61: Puntos visibles del mapa 2D.

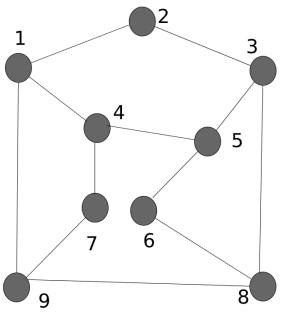


Figura 28.62: Grafo de los puntos visibles.

El movimiento de un actor se desarrolla como el desplazamiento entre puntos visibles. Este movimiento puede generalizarse como el desplazamiento por los nodos de un grafo, y de esta forma definimos el problema de la construcción de un camino desde un punto visible a otro punto visible como un problema de encontrar el camino entre dos nodos de un grafo.

28.4.2. Problema de búsqueda en grafos

Un grafo es una estructura que consta de nodos y arcos. En el siguiente código se define una clase para representar un grafo.

Listado 28.19: class Grafo

```
class Grafo:
    def __init__(self,tipo='Dir',nombre='Noname'):
        self.g={}
        self.tipo=tipo
        self.id=nombre
        self.NumNodos=0
        self.NumArc=0

    def addNodo(self,Nodo):
        if not Nodo in self.g.keys():
            self.g[Nodo]=[]

    def addArco(self,NodoOrigen,NodoDestino,InfArco):
        self.addNodo(NodoOrigen)
        self.addNodo(NodoDestino)
        self.g[NodoOrigen].append((InfArco,NodoDestino))
        if not (self.tipo=='Dir'):
            self.g[NodoDestino].append((InfArco,NodoOrigen))

    def nodosSucesores(self,Nodo):
        return self.g[Nodo]
```

Las operaciones básicas definidas son: **añadir un arco**,un **nodo** y obtener los **sucesores** de un nodo. Según sea el acceso de un nodo a otro, encontraremos nodos donde solo es posible en una dirección que denominaremos **grafos dirigidos**; en caso contrario, tendremos **grafos no dirigidos**. Esta diferencia la introducimos en la clase grafo mediante el valor tipo de la clase, que por defecto será dirigido.

Definido la estructura de grafo, podemos construir un problema como se comentó en 28.3.1 para lo que definimos la clase ProblemaGrafo.

Listado 28.20: class ProblemaGrafo

```
class ProblemaGrafo(Problema):
       def __init__(self,Grafo,NodoInicial,NodoFinal):
            Problema.__init__(self,NodoInicial,"Grafo")
            self.estadoFin=NodoFinal
            self.graf=Grafo
        def estadoValido(self,Nodo):
            return Nodo in self.graf.g.keys()
        def sucesor(self,Nodo):
            return self.graf.nodosSucesores(Nodo)
        def meta(self,Nodo):
            return Nodo==self.estadoFin
        def estadoInit(self):
            return self.estInit
```

Esto proporciona un método eficiente de construir el camino por el grafo, desde un nodo origen a un nodo destino.

Obtenida la secuencia de puntos visibles por donde pasará el actor,el siguiente proceso será el desplazamiento físico entre los puntos. Este proceso puede ser simple, si no hay elementos que puedan interaccionar con las trayectorias elegidas, o por contra, puede ser más complejo si podemos ser interceptados o necesitamos modificar en un momento dado la trayectoria planificada. Estos problemas quedan fuera del alcance de esta sección, así que sólo nos centraremos en encontrar las trayectorias correctas.

La construcción del camino a seguir, se realiza obteniendo la solución al problema de ir de un nodo (punto visible) de un grafo a otro nodo destino. La solución es la secuencia de punto visibles que ha de ir ocupando el agente consecutivamente.

Una forma de mejorar los resultados obtenidos mediante la simple búsqueda de soluciones es considerar un tipo específico de grafo, el grafo de punto visibles, y explotar una serie de características propias de la localización de sus nodos.

La primera de estas característica es que un punto visible P viene definido por unas coordenadas. En un mundo de dos dimensiones $P = (x, y)$ serán dichas coordenadas, mientras que en un escenario en 3 dimensiones vendrá definido por $P = (x, y, z)$. Esta caracteristica permite plantearse, no sólo encontrar el camino que una dos puntos visibles; si no que ese **camino sea bueno**. En un supuesto donde el desplazamiento consume energía del agente (muy generalizado), un buen camino será aquel que menos recorrido tenga que realizar.

El cambio de un punto visible P hacia otro punto visible P' provocará en un actor un desgaste equivalente a trasladarse en linea recta desde P a P'. Este gasto en términos de posiciones será la distancia en línea recta desde P a P', esto es:

$$d(P, P') = d((x, y), (x', y')) = \sqrt{(x - x')^2 - (y - y')^2} \tag{28.5}$$

Esto establece que la acción de ir de un nodo a otro incurre en un gasto cuantificado por la diferencia de posición (ecuación 28.5), por lo que se pude incorporar a las estrategias de búsqueda informadas 28.3.3. La acción (arco) entre el nodo P y nodo P' vendrá definida como (P->P',d(P,P')), incorporando el valor del costo del nodo P' como la suma de todas las distancias recorridas. En definitiva, el total de distancia recorrida por el agente. De esta forma podremos utilizar una estrategia de costo uniforme.

Otra de las cuestiones que se puede explotar es el conocimiento aproximado de la distancia que nos queda por recorrer. Esta distancia sería la existen desde el nodo **i** al nodo destino final **f**, luego es factible utilizar la distancia que falta por recorrer como valor heurístico para la búsqueda informada con una estrategia A*. Si el movimiento estuviese limitado a los 4 desplazamientos básicos: arriba, abajo, izquierda y derecha, otro aproximación más fiel a la distancia que falta por recorrer es la distancia en vertical más la distancia en horizontal que separa el nodo del destino. Esta nueva valoración se denomina **distancia manhattan** y su expresión viene definida por la la ecuación 28.6. En la figura 28.63 se puede observar el significado geométrico de cada una de las heurísticas expresadas.

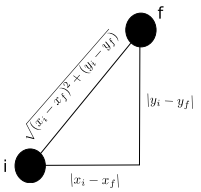


Figura 28.63: Heurísticas para la estrategia A*.

$$dm(P_i, P_f) = |x_i - x_f| + |y_i - y_f| \tag{28.6}$$

La incorporación de la heurística al problema del grafo se muentra en el siguiente código.

Listado 28.21: class Grafo con información

```
import map

class GrafoMapa(Grafo):
    def __init__(self,map,h='EUCLIDEA'):
        self.dato=map
        self.id="Mapa"
        self.h=h
    def addNodo(self,Nodo):
        pass
    def addArco(self,NodoOrigen,NodoDestinod,InfArco):
        pass
    def nodosSucesores(self,Nodo):
        f,c=Nodo
        mv=[]
        anchura,altura=self.dato.tablero
        if (f<(altura-1)) and (not self.dato.mapa[f+1][c]==1):
            mv.append(({'Acc':'AB','val':1},(f+1,c)))
        if (f>0) and (not self.dato.mapa[f-1][c]==1):
            mv.append(({'Acc':'AR','val':1},(f-1,c)))
        if (c<(anchura-1)) and (not self.dato.mapa[f][c+1]==1):
            mv.append(({'Acc':'DE','val':1},(f,c+1)))
        if (c>0) and (not self.dato.mapa[f][c-1]==1):
            mv.append(({'Acc':'IZ','val':1},(f,c-1)))
        return mv

    def heuristica(self,nodo1,nodo2):
        y1,x1=nodo1
        y2,x2=nodo2
        s=0.0
        if self.h=='EUCLIDEA':
            s= ((x1-x2)**2.0+(y1-y2)**2.0)**(1.0/2.0)
        elif self.h=='CUNIFORME':
            s=0.0
        else:
            s= abs(x2-x1)+abs(y2-y1)
        print "H(%s,%s)=%d"%(str(nodo1),str(nodo2),s)
        return s
```

Hasta este punto hemos solucionado el problema de encontrar un camino, incluso el mejor, entre un punto visible y otro de nuestro mapa. Siendo capaces de construir o generar el grafo de los puntos visibles, ya tenemos solucionado el problema del movimiento de los actores en el escenario o mapa.

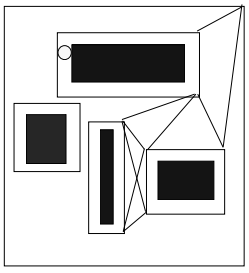


Figura 28.64: Construcción automática del grafo de puntos visibles.

28.4.3. Obtención del grafo de puntos visibles

La primera opción para la obtención del grafo de los puntos visibles es de forma manual. Dado un mapa se señalan en él los puntos y se unen para generar el grafo. Esto obliga a construir un editor de grafos de puntos visibles y adjuntar a cada mapa su grafo correspondiente.

Una alternativa a la construcción manual es construir un grafo de forma automática. El primer método parte de un estructura particular de un mapa. Partamos de un mapa construido con figuras geométricas definidas por unos vertices y unas arístas. Consideremos las figuras que configuran los obstáculos o paredes de nuestro mapa, y extendamos su geometría una distancia igual al tamaño de nuestro agente (figura 28.64.

En estos momentos tendremos un nuevo conjunto de vertices que representarán los puntos visibles de nuestro mapa. Para terminar de construir el grafo, se añadirán las aristas desde un vertice a todos a quellos que tiene visión directa.

De esta forma construimos un grafo deforma automática para el desplazamiento de los agentes en el mapa. Hemos de destacar que los grafo de puntos visibles generados de esta forma originan un movimiento de los agentes siguiendo la geometría de los objetos como obstáculos o paredes.

28.4.4. Aumentando la resolución

Con los GPV (grafos de puntos visibles), obtenidos se tiene un conjunto no muy numeroso de putos visibles y con una distribución no muy uniforme por todo el mapa. Para aumentar el nivel de realismo en el movimiento de los agentes se tendra que aumentar el número de puntos visibles, y que estos estén distribuidos por el mapa de una forma uniforme. En un principio si consideramos una distancia mínima **dm** para separar puntos visibles, tendríamos una forma de ajustar el realismo o nuestra resolución mediante el ajuste del valor de dm.

Si además consideramos que los movimientos que poden realizar los agentes están acotados, tendremos un nuevo mecanismo de generar un grafo de puntos visibles. Este nuevo mecanismo es el siguiente:

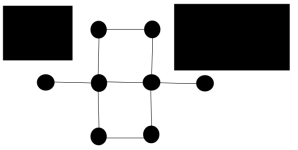


Figura 28.65: Primeros pasos de la creación de un grid.

1. Colocar un punto visible en una posición del mapa.
2. Añadir este punto visible a una lista de PV no visitados.
3. Mientras tengamos punto visibles en la lista de no visitados
4. sacar un punto p
5. añadir a la lista de PV no vistados tantos nuevos puntos como movimientos posibles y distancia dm que no hayamos visitado.
6. añadir el punto analizado a la lista de PV visitados

Figura 28.66: Tiles utilizados.

En el siguiente código se ha creado un clase mapa, que permite definir un escenario basado en tiles, donde podemos fijar una casilla inicial, muros y una casilla final. Construido el mapa podemos definir un problema de búsqueda en grafo para el grid creado y ver como se comporta los algoritmos para determinar el camino.

Un ejemplo con las primeras iteraciones se muestra en la figura 28.65.

Un caso particular de esta técnica es cuando el propio mapa está diseñado mediante pequeñas celdas o **tiles**.En estos caso el propio escenario es ya el grafo de puntos visbles, y podemos asignar a cada una de las celdas accesibles un punto visible y los movimientos a las celdas adyacentes como los arcos a nuevos puntos visbles.

Listado 28.22: class Grafo con información

```
1  import sys,pygame,csv
2
3  class Map:
4      def __init__(self,celda=(20,20),tablero=(50,30),m=None):
5          self.celda= 20,20
6          self.tablero= 50,30
7          self.size = width,height = tablero[0]*celda[0],tablero[1]*
               celda[1]
8          self.s = pygame.display.set_mode(self.size)
9          loseta=self.CreaCelda((113,132,132))
10         muro=self.CreaCelda((255,181,66))
11         inicio=self.CreaCelda((255,0,0))
12         fin=self.CreaCelda((0,0,255))
13         self.elementos=[loseta,muro,inicio,fin]
```

```
14          if m==None:
15              self.CreaMap(self.tablero[0],self.tablero[1])
16          else:
17              self.mapa=m
18
19      def CreaCelda(self,color):
20          s=pygame.Surface((self.celda[0],self.celda[1]),0,16)
21          s.fill(color)
22          pygame.draw.rect(s,(255,181,66),pygame.Rect(0,0,20,20),2)
23          return s
24
25      def AnadeCelda(self,color):
26          self.elementos.append(self.CreaCelda(color))
27
28      def CreaMap(self,w,h):
29          self.mapa=[]
30          for y in range(h):
31              self.mapa.append([])
32              for x in range(w):
33                  self.mapa[y].append(0)
34
35      def  PintaMapa(self):
36          for y in range(self.tablero[1]):
37              for x in range(self.tablero[0]):
38                  rect=pygame.Rect(x*self.celda[0],y*self.celda[1],
                        self.celda[0],self.celda[1])
39                  self.s.blit(self.elementos[self.mapa[y][x]],rect)
40
41      def ModificaMapa(self,celdaIndx,infor):
42          titulo=pygame.display.get_caption()
43          pygame.display.set_caption(infor)
44          NoFin=True
45          while NoFin:
46              for event in pygame.event.get():
47                  if  event.type == pygame.KEYDOWN and event.key ==
                        pygame.K_ESCAPE:
48                      NoFin=False
49              pos=pygame.mouse.get_pos()
50              pos=pos[0]/self.celda[0],pos[1]/self.celda[1]
51              b1,b2,b3= pygame.mouse.get_pressed()
52              rect=pygame.Rect(pos[0]*self.celda[0],pos[1]*self.celda
                    [1],self.celda[0],self.celda[1])
53              if b1:
54                  self.s.blit(self.elementos[celdaIndx],rect)
55                  self.mapa[pos[1]][pos[0]]=celdaIndx
56              elif b3:
57                  self.s.blit(self.elementos[0],rect)
58                  self.mapa[pos[1]][pos[0]]=0
59              pygame.display.flip()
60          pygame.display.set_caption(titulo[0])
61
62      def SalvaMapa(self,fn):
63          f=csv.writer(open(fn,"wr"),delimiter=',')
64          for l in self.mapa:
65              f.writerow(l)
66
67      def CargaMapa(self,fn):
68          f=csv.reader(open(fn,'rb'),delimiter=',')
69          self.mapa=[]
70          for r in f:
71              self.mapa.append([int(x) for x in r])
72      def DefMuros(self):
73          self.ModificaMapa(1,'Definir Muros')
74      def Entrada(self):
75          self.ModificaMapa(2,'Definir Entrada')
76      def Salida(self):
77          self.ModificaMapa(3,'Defir Salida')
```

En la figura 28.67 podemos ver un ejemplo de un mapa construido con la clase mapa y el conjunto de tiles de la figura 28.66.

Finalmente, podemos observar el camino encontrado por la estrategia A* y una heurística de distancia euclídea en la figura 28.68

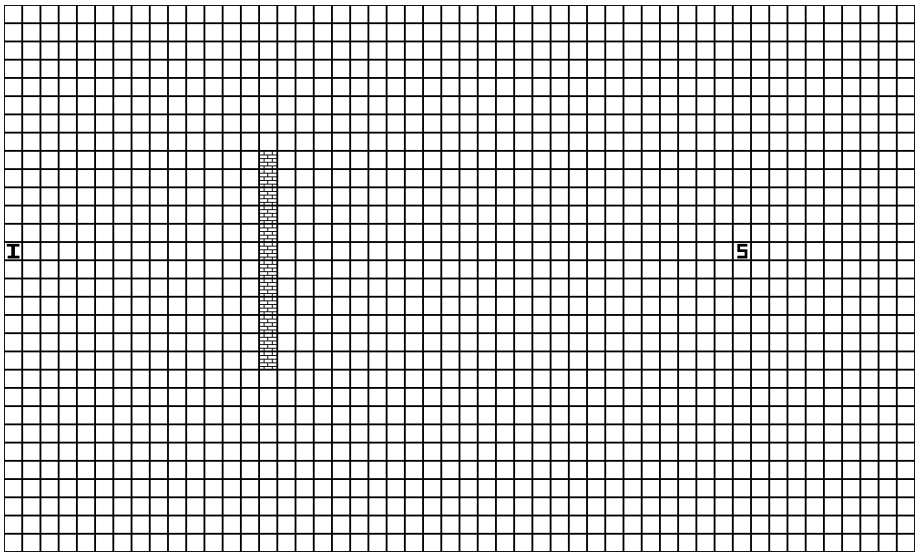


Figura 28.67: Mapa 2D

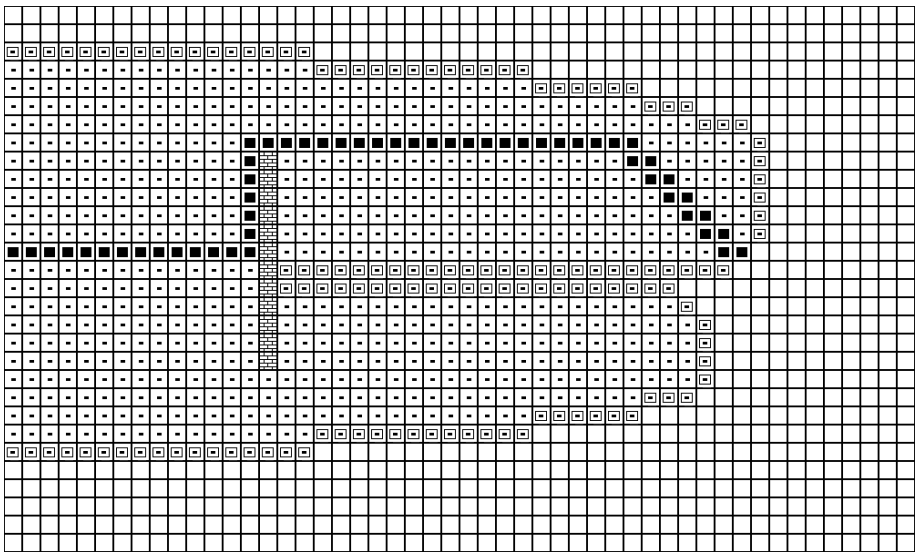


Figura 28.68: Camino con la heurística de la distancia euclídea.

28.5. Diseño basado en agentes

Como ya se ha comentado en la Sección 28.1, en los últimos años ha habido un creciente uso de tecnología relacionada con la IA en la construcción de videojuegos. Este uso ha sido enfocado principalmente desde la perspectiva del desarrollo de agentes inteligentes que hacen uso de algoritmos de IA para chasing (persecución y escape), para búsqueda de rutas, para planificación, para decidir... Cada vez son más los creadores de videjuegos (p.e. Sid Meier (Civilization), Chris Crawford (Balance of Power) y Peter Molyneux (Populous) por citar algunos) que emplean estos agentes inteligentes con capacidad de decisión, para crear en sus juegos personajes virtuales que se comporten de una forma parecida a como lo hace un humano. Se consigue así que los jugadores piensen que se están enfrentando a oponentes reales.

Figura 28.69: Personajes Virtuales dotados con comportamientos similares a los de un personaje real.

En la mayoría de las ocasiones, en los videojuegos, los agentes no son desarrollados de manera independiente sino como entidades que constituyen un sistema más completo. De este modo, los videojuegos se están convirtiendo en sistemas multiagente, a partir de ahora se hará referencia a ellos como MAS (Multi-Agent Systems). En estos sistemas existirán un conjunto de agentes que son empleados para simular personalidades y caracteres diferentes para los personajes de los juegos y que pueden coordinar sus comportamientos de grupo y colaborar para lograr un fin (p.e. matar, o siendo menos drásticos, hacer prisionero al jugador).

En esta sección se mostrará como diseñar agentes que puedan ser empleados en videojuegos, siendo esta última restricción importante ya que no se va a pretender crear agentes que emulen la inteligencia humana, sino que el objetivo será crear agentes que hagan que el usuario **crea** que el videojuego tiene inteligencia, aunque no sea así. Se presentarán dos enfoques distintos de diseño de agentes uno basado en **comportamientos** y otro basado en **objetivos**, prestando mayor atención al primero por ser el más usual. Para ello primero se dará una definición del concepto de agente, y una posible clasificación de los mismos. Después se presentarán los tipos de agente que se suelen construir en los videojuegos. Se finalizará la sección con una discusión sobre las implicaciones que tendría construir sistemas multiagentes completos y unas breves reflexiones sobre hacia donde puede tender el uso de agentes y sistemas multiagentes en el campo de los videojuegos y los retos que hay abiertos.

Figura 28.70: Desde que el Profesor Nick Jennings enunciara la frase "*Los agentes constituyen el próximo avance más significativo en el desarrollo de sistemas y pueden ser considerados como la nueva revolución en el software*", muchos son los que se han subido al carro indicando que sus aplicaciones son agentes. Los agentes están de moda.

28.5.1. ¿Qué es un agente?

No existe una definición plenamente aceptada por la comunidad científica del concepto agente inteligente, siendo la más simple la de Russell y Norvig [79], que fue presentada en la sección 28.1.3 y que conviene recordar aquí: "*un agente es una entidad que percibe y actúa sobre un entorno*" (vea la figura 28.7). En términos matemáticos esta definición lleva a pensar en un agente como una función (el comportamiento) que proyecta percepciones en acciones. Cabe notar, que esta definición es tan amplia, que permite que a numerosos sistemas software se les pueda asignar la etiqueta de agente, por otra parte hay que destacar que existe un gran interés tanto académico como industrial en ello (vea la figura 28.70).

Para delimitar un poco más el concepto de agente, comienzan a aparecer en la literatura una serie de calificativos cuyo objetivo es denotar aquellas propiedades que debe cumplir un agente. De este modo Wooldridge y Jennings [104] definen a un agente como un *sistema capaz de actuar de forma autónoma y flexible en un entorno*, donde flexible es entendido como:

- **Reactivo**, con capacidad de respuesta a cambios en el entorno donde está situado.

- **Proactivo**, con capacidad de decidir cómo comportarse para cumplir sus planes u objetivos.
- **Social**, con capacidad de comunicación con otros agentes.

Así, para que un software pudiera ser considerado un "agente" debería de tener las características antes expuestas. No obstante, a lo largo de los años, los investigadores del campo han usado nuevas características para calificar a los agentes, a continuación se citan algunas de las recopiladas por Franklin y Graesser [36]:

- **Continuidad Temporal**, con capacidad para ejecutarse ininterrumpidamente.
- **Racionalidad**, con capacidad de elegir siempre la acción "correcta" en función de lo percibido en el entorno.
- **Adaptabilidad**, con capacidad de cambiar su comportamiento basándose en el aprendizaje que realice.
- **Movilidad**, con capacidad para trasladarse a través de una red telemática.

Otras características que se asumen como propiedades de los agentes son la **Veracidad**, por la que se se supone que un agente no va a proporcionar información falsa a propósito y la **Benevolencia**, que implica que un agente esté dispuesto a ayudar a otros agentes, si esto no entra en conflicto con sus intereses.

Llegados a este punto, cabe entonces hacerse la siguiente pregunta, "¿qué características debe poseer un sistema para poderle asignar el calificativo de agente?" Al respecto, se debe indicar que no existe un concenso sobre cual es la importancia de cada una de las propiedades que se han enunciado para definir un agente. Sin embargo, si se puede afirmar que son la existencia de estas, en mayor o menor medida, las que distinguen agentes de meros programas informáticos (vea figura 28.71). Desde la perspectiva que nos ocupa, el diseño de videojuegos, cabe otra pregunta, "¿qué características deben poseer los agentes usados en los videojuegos?"

Figura 28.71: El profesor Van Dyke Parunak compara un agente con las navajas suizas, la navaja corresponde a la definición básica de agente, y cada accesorio de la navaja con cada una de las propiedades del agente, que son usados solo si se necesitan.

En este sentido a la respuesta antes dada, hay que añadirle un matiz, y este es el siguiente: Mientras en los agentes del campo de la IA académica se pueden emplear máquinas especiales con gran poder de cómputo para ejecutarse (p.e. la famosa deep blue), los agentes diseñados para juegos tienen su poder de cómputo limitado a máquinas más normales y es mucho menor, y además compartido con otros procesos intensivos en CPU, como por ejemplo la simulación física, la generación de los gráficos y el tráfico de red. Esto conduce al desarrollo de agentes más limitados en características o en el despliegue de las mismas, pero que den sensación de inteligencia.

Una de las características que se han presentado como interesante, y que por otra parte es resaltada como necesaria en los agentes cuando se le añade el calificativo de "inteligente", es la de *racionalidad*. Como ya se ha comentado, esta característica implica que el agente haga siempre lo correcto, pero hay que preguntarse ¿qué es lo correcto? Una primera aproximación, y por otra parte usual, es la de definir lo correcto como aquello que le permite al agente obtener un resultado mejor, por lo tanto habrá que medir el éxito. Para ello, es preferible emplear medidas de rendimiento o utilidad, que estén asociadas al entorno, más que de desempeño asociadas a cómo de bien lo hace el agente. Es conveniente resaltar en este punto que no existe una medida universal para todo agente, cada agente tendrá su propia medida y ésta será dependiente del problema.

Agente y Problema

Los agentes racionales representan soluciones a problemas.

Un agente racional deberá emprender en cada momento la acción que en principio maximice su medida de desempeño o rendimiento, basándose en las evidencias aportadas por la secuencia de percepciones de dicho momento y en el conocimiento que el agente mantiene de momentos anteriores. De este modo, la racionalidad de un agente en un instante determinado vendrá dada por:

- El valor de la medida de desempeño, rendimiento o utilidad del agente.
- El conocimiento acumulado por el agente sobre el entorno en el que desempeña su función.
- El conjunto de acciones que puede realizar en ese instante.
- La secuencia de percepciones que ha captado haste ese instante.

En el caso de los videojuegos, un agente racional debería tener una medida de rendimiento asociado al entorno que permitiera guiar su comportamiento. Por ejemplo, en los juegos de acción en primera persona (FPS), los agentes podrían ser empleados para simular comportamientos de adversarios y la medida de rendimiento será inversamente propocional al nivel de *stamina* que tiene el jugador tras ejecutar la acción. La elección de la siguiente acción se podría realizar en base a la capacidad para reducir este valor, o considerando de las que ya ha ejecutado, aquella que mejor resultado le dio.

28.5.2. ¿Cómo se construye un agente?

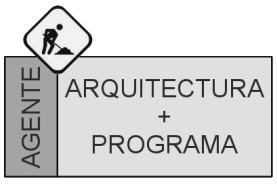


Figura 28.72: La relación entre agentes, arquitectura y programas podría resumirse de la siguiente manera: Agente = arquitectura + programa. La construcción de un agente implica desarrollar un programa y desplegarlo sobre una arquitectura.

Un agente es un programa junto con una arquitectura (vea la figura 28.72). El programa implementará una función que transforma secuencias de percepciones en acciones. La arquitectura define los mecanismos que permiten interconectar los componentes, tanto de software como de hardware, es decir establecen las relaciones que fluyen entre las entradas (sensores), las salidas (actuadores) y el razonamiento interno del agente. La IA permite crear programas de agentes para el razonamiento interno.

En la construcción de agentes, la complejidad del diseño de los mismos vendrá determinada, a parte de por su función (o comportamiento), complejidad de las percepciones, y la medida de desempeño, rendimiento o utilidad empleada para medir como de correcto es su comportamiento, por las características del entorno en el que tenga que llevarla a cabo. Los agentes son sistemas que toman decisiones empotrados en entornos, su conocimiento es por tanto necesario para la construcción eficiente de agentes. Tradicionalmente, en el campo de la IA se han establecido los siguientes tipos de entornos:

- **Totalmente observable / Parcialmente observable**. Será un entorno totalmente observable aquel en el que se tiene acceso a todos los aspectos medibles necesarios para la toma de decisión. Por contra un entorno parcialmente observable no permite captar todo el entorno. En aplicaciones reales, este último tipo de entorno es el más frecuente. En términos de rendimiento el primero es el más recomendable y conveniente.
- **Determinista / Estocástico**. Un entorno es determinista cuando el siguiente estado que se alcanza se puede determinar a partir del estado actual y la acción llevada a cabo, no hay incertidumbre. En un entorno estocástico no siempre se puede determinar con precisión el estado que se alcanza desde el estado actual y la acción del agente. Los entornos reales generalmente son estocásticos, siendo lo preferible tratar con entornos deterministas.

- **Episódico / Secuencial**. En los entornos episódicos un agente divide su función en episodios atómicos que se suceden uno tras otro, no hay dependencia con las acciones anteriores. Cada episodio es una percepción más su acción asociada. En los entornos secuenciales las decisiones y acciones presentes pueden afectar a las decisiones y acciones futuras.

- **Estático / Dinámico**. En el entorno estático, las condiciones no puede cambiar mientras el agente está deliberando, este tipo de entornos son los más fáciles de tratar por no requerir estar pendiente de los cambios que suceden. En los entornos dinámicos, estos pueden cambiar mientras el agente decide que hacer, y puede ocurrir que este tome una decisión sin saber realmente como está el entorno. Los más complejos son los entornos dinámicos.

- **Discreto / Continuo**. Los entornos discretos son aquellos en los que es posible distinguir y establecer un conjunto finito de estados posibles. El caso contrario es un entorno continuo.

El entorno, o mejor dicho la complejidad del mismo, también se ve afectada por la cantidad de agentes que intervienen en él. De este modo se pueden encontrar desde entornos con un único agente o con un conjunto de agentes que no colaboran, a entornos con multitud de agentes que colaboran y forman sistemas multiagente.

Cuanto más simple es el entorno, más llevadera será la tarea de construcción del agente o agentes. En el caso de los videojuegos, y puesto que son entornos controlables, se tiende a trabajar simplificando al máximo el entorno, siendo lo habitual entornos totalmente observables, deterministas y discretos.

La estructura de un programa agente simple, en pseudocódigo, es mostrada en la figura 28.73. A continuación se estudiarán varios tipos de agente en función del programa y arquitectura que se emplean para su construcción:

- **Agente de Razonamiento Simbólico.** Emplean representaciones símbolicas del entorno y de la conducta que se desea que tenga, toda esta representación es manipulada sintácticamente. Este tipo de agentes toman decisiones sobre que es lo que tienen que hacer por medio de manipulaciones simbólicas. El ejemplo típico es el del agente que utiliza la lógica para representar el entorno y el razonamiento lógico explícito para decidir que hacer.

  Este tipo de agentes requieren una traducción precisa y adecuada del entorno en una descripción simbólica, lo cual puede ser complejo. Además su coste en tiempo para obtener resultados es también alto, puesto que los algoritmos de manipulación simbólica tienen una complejidad alta. Esto los hace poco aplicables en el caso de los videojuegos.

- **Agente Reactivo Simple (o de Reflejo Simple).** Puede considerarse el programa de agente más sencillo. Su funcionamiento se basa en seleccionar la acción sólo sobre las percepciones actuales del agente, ignorando las percepciones históricas, para ello almacena asociaciones entrada/salida frecuentes en forma de reglas condición-acción: Si <Percepción>entonces <acción>.

  Este tipo de agentes cuentan con una inteligencia limitada y trabajan bien en entornos totalmente observables, lo cual los hace poco útiles en problemas reales medianamente complejos pero si pueden tener utilidad en el mundo de los videojuegos: *Si* detectas un obstáculo *entonces* cambia de dirección.

- **Agente Reactivo basado en modelo.** En gran cantidad de ocasiones un agente no podrá tomar una decisión teniendo en cuenta una única percepción, porque esta no proporciona toda la información necesaria, es por ello necesario emplear estados que de alguna forma guarden información sobre las percepciones históricas o que ya no son observables. Además se necesita información sobre cómo evoluciona el mundo, independiente del agente.

  En este tipo de agentes la percepción actual se interpreta a partir del estado anterior utilizando información sobre: cómo evoluciona el entorno independientemente del agente y la influencia en el entorno de las acciones del agente. Los modelos serán empleados para representar cómo evoluciona el entorno.

- **Agente Reactivo basado en subsunción.** En este tipo de agentes la toma de decisión (la acción) se realiza mediante un conjunto de módulos de comportamiento que realizan tareas. Los comportamientos se agrupan en capas que están organizadas en orden decreciente de priodidad. De este modo los comportamientos de las capas inferiores tendrán más prioridad que las de las capas superiores. No se pueden ejecutar módulos de distintas capas, los de las capas inferiores inhiben a los de las capas superiores. Es decir, cada percepción puede acarrear la ejecución de varios módulos (al satisfacer las condiciones de ejecución de los mismos), se deberá entonces comprobar que comportamientos no son inhibidos por otros y llevarlos a cabo.

- **Agente Deliberativo.** Son una mejora sobre los Agentes de Razonamiento Simbólico, y surgen porque con los estados no es suficiente para tomar una decisión, ya que ésta muchas veces depende de cual es la misión del agente. Por tanto se requiere información sobre el objetivo o meta del agente. Estos agentes se caracterizan por la utilización de modelos de representación simbólica del conocimiento (tanto del entorno como de la meta) y suelen basarse en la teoría clásica de planificación: Parten de un estado inicial y emplean un sistema de planificación que genera un conjunto de pasos (un plan) para lograr el objetivo para el cual fueron diseñados. Por tanto, la búsqueda y la planificación son muy importantes para la construcción de este tipo de agentes.

  La arquitectura deliberativa más estudiada y extendida es la arquitectura BDI (Belief, Desire, Intention). Esta se caracteriza por el hecho de que los agentes que la implementan están dotados de los estados mentales de Creencias (modelo del mundo), Deseos (metas) e Intenciones (plan de acción), y para definir los planes se tienen en cuenta sus creencias y deseos. Su principal problema a la hora de aplicarse es que requieren un elevado tiempo de respuesta, esto lo hace especialmente poco útiles en el caso de los videojuegos. No obstante, recientemente desde el campo académico se están proponiendo formas de usar la arquitectura BDI para modelar comportamientos de personajes en videojuegos, por ejemplo algunas pruebas se han realizado en el videojuego Quake 2.

  En ocasiones, las metas por si solas tampoco son suficientes, ya que no basta con alcanzar una meta sino hacerlo de la mejor forma posible. Para garantizar la selección de la mejor meta es necesario una medida que permita comparar decisiones o planes. Esta medida se denomina utilidad. Una acción puede tener más utilidad que otra, el agente debe ir buscando aquellas que les proporcionen más utilidad.

- **Agente Híbrido.** Este tipo de agentes combinan componentes de tipo reactivo y componentes de tipo deliberativo. Usualmente, la parte reaactiva se encarga de reaccionar en función de lo que ocurra en el entorno sin dedicar demasiado tiempo a razonar y la parte deliberativa es la que se encarga de planificar a más largo plazo y a soportar los procesos de toma de decisión.

```
function ESQUELETO-AGENTE (percepciones) returns accion
   static: memoria /*La memoria del agente del entorno*/

   memoria  ←  ACTUALIZAR-MEMORIA(memoria, percepciones);
   accion   ←  ELEGIR-MEJOR-ACCION(memoria);
   memoria  ←  ACTUALIZAR-MEMORIA(memoria, percepciones);
   return accion ;
```

Figura 28.73: Visión abstracta del funcionamiento de un agente.

28.5.3. Los comportamientos como guía de diseño

Los agentes pueden ser programados en base a sus comportamientos. Un comportamiento especifica las tareas o servicios que realiza un agente para lograr sus objetivos. Cada comportamiento puede realizar una tarea simple, que en el caso de los videojuegos pueden ser, "Lanzar una pregunta" o "Resoplar" aunque también se pueden crear comportamientos más complejos como por ejemplo "Huir" o "Atacar".

Para realizar una programación basada en comportamientos se deben seguir los siguientes pasos (vea figura 28.74):

- Estudiar el entorno.
- Analizar que es lo qué debe ser capaz de hacer el agente. Estudiar cuales son los comportamientos deseados y cuando ocurren en función del entorno.
- Asociar a cada comportamiento una funcionalidad. Establecer que es lo que debe ocurrir como acción asociada a cada comportamiento.
- Relacionar los comportamientos. Establecer las conexiones que existen entre los comportamientos, es decir como se conectan y cuando ocurrren.

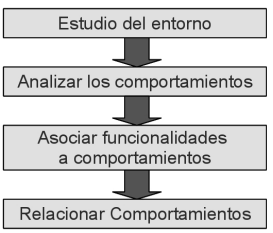


Figura 28.74: Pasos en el diseño de agentes empleando los comportamientos como guía de diseño.

El agente reactivo, en alguna de sus variantes, es el más fácil de construir para programar agentes basados en comportamientos, ya que realiza tareas sencillas y de una forma rápida. Es por esto, por lo que resulta más útil en el campo de los videojuegos. El esquema general del programa de agente reactivo es el que se muestra en la figura 28.75.

Como se puede observar su funcionamiento se basa en una recepción de percepciones del entorno (proporcionados por los sensores) que al ser interpretada produce un cambio en el estado interno del agente, lo cual lleva asociada la selección del procedimiento o rutina con la que se debe reaccionar en ese estado a esa percepcion, y por último se produce la reacción propiamente dicha. Ésta consiste en la ejecución del procedimiento o rutina. En este esquema de funcionamiento no hay ningún mecanismo explícito de representación del conocimiento ni hay procesos de manipulación sobre el mismo, es esto lo que simplifica el diseño y construcción del agente, así como su uso con tiempos de respuesta inmediatos.

```
function ESQUELETO-AGENTE-REACTIVO (percepciones) returns accion
   static: reglas

   estado   ← INTERPRETAR(percepciones;
   regla    ← ELEGIR-REGLA(estado,reglas);
   accion   ← DISPARO(regla);
   return accion;
```

Figura 28.75: Visión abstracta del funcionamiento de un agente reactivo.

Este tipo de agente se suele diseñar de forma parecida a los sistemas basados en reglas, en los que existe una base de reglas y un motor de inferencia. Cada regla enlaza una percepción con una acción y el sistema de inferencia determina que regla ejecutar en función de la percepción recibida.

Los agentes reactivos más usados en el campo de los videojuegos son los agentes reactivos basados en modelos. Para el diseño de este tipo de agentes se emplean autómatas de estados finitos (en la Sección 28.1.4 ya se introdujo la idea). Se divide el comportamiento general del objeto o personaje virtual del videojuego en un número finito de estados, cada uno de ellos corresponde a una situación del entorno, que tendrá asociado un comportamiento particular del personaje y se establecen las transiciones que pueden ocurrir entre ellos y las condiciones o reglas que se deben cumplir para pasar de un estado a otro de una manera determinista, lo cual implica que sea predecible el comportamiento.

En la figura 28.76 se muestra cual es el esquema de funcionamiento del agente reactivo basado en un atómata de estados finitos (FSM (Finite State Machine)). El funcionamiento de este agente se basa en a partir del estado actual, almacenado en la memoria, comprobar que regla de las posibles reglas que le permiten cambiar de estado puede ser disparada en función de las percepciones del entorno recibidas. El disparo de esa regla le permite cambiar de estado y ejecutar la acción asociada al nuevo estado que se alcance. Una vez ejecutada esta acción hay que convertir el estado alcanzado como estado actual. Conviene notar que solo será posible disparar una única regla y que por lo tanto en cada instante el autómata se encontrará en un único estado.

La elección de los autómatas como arquitectura de funcionamiento se debe a que:

- **Son simples y rápidos de codificar.** Existen muchas formas de programar un autómata todas ellas razonablemente simples de programar.
- **Son fáciles de depurar.** Al estar formados por estados es fácil de depurar ya que se podría añadir codigo de depuración en cada uno de ellos.
- **Tienen poca carga computacional.** No poseen procesos computacionales complejos, su funcionamiento es muy simple.
- **Son intuitivos.** Se asemeja a la forma de pensar o actuar de los humanos.
- **Son flexibles.** Pueden ser fácilmente ajustados por el programador para proporcionar el comportamiento requerido por el diseñador del juego.

```
function ESQUELETO-AGENTE-REACTIVO-FMS (percepcion) returns accion
   static: reglas, estados
   dinamic: memoria

   regla    ← ELEGIR-REGLA(percepcion,memoria,reglas);
   estado   ← REALIZAR-CAMBIO-ESTADO(memoria,regla);
   accion   ← EJECUTAR-ACCION(estado);
   memoria  ← ACTUALIZAR-ESTADO(estado);
   return accion;
```

Figura 28.76: Visión abstracta del funcionamiento de un agente reactivo basado en un atómata de estados finitos (FSM).

No es objetivo de esta sección dar una definición formal del autómata, para el propósito de este curso basta con saber que es un modelo de comportamiento de un sistema o un objeto complejo y que se compone de estados, eventos y acciones. Un estado representa una situación. Un evento es un suceso que provoca el cambio de un estado a otro. Una acción es un tipo de comportamiento.

No hay que olvidar que en un instante el autómata solo podrá estar en un único estado.

En realidad, lo que se usa en los videojuegos para modelizar comportamientos no son autómatas de estados finitos, ya que la finalidad de estos autómatas es la de reconocer lenguajes regulares (los lenguajes formales más simples según la Clasificación de Chomsky). Este tipo de autómatas no poseen salida asociada. Lo único que ofrecen es una decisión sobre si una cadena, formada por elementos tomados de un alfabeto, dada como entrada, pertenece o no al lenguaje regular que el autómata reconoce.

El funcionamiento de este tipo de autómatas está dirigido por una función de transición, que a partir de un estado y un elemento de la cadena que se está analizando, desplaza a otro estado del autómata. Si tras leer todos los caracteres a partir del estado inicial del autómata se detiene en un estado final o de aceptación, entonces la cadena es del lenguaje en caso contrario no lo es (vea la figura 28.77).

En los videojuegos se usarán dos tipos especiales de autómatas finitos para modelizar comportamientos, las **máquinas de Moore** y las **máquinas de Mealy**. Estas máquinas o autómatas intentan aumentar la expresividad de los autómatas finitos deterministas generando una salida en función de su estado actual y de la entrada. La máquina de Moore tendrá asociada las salidas al estado y las de Mealy las tendrán asociadas a las transiciones (vea la figura 28.78).

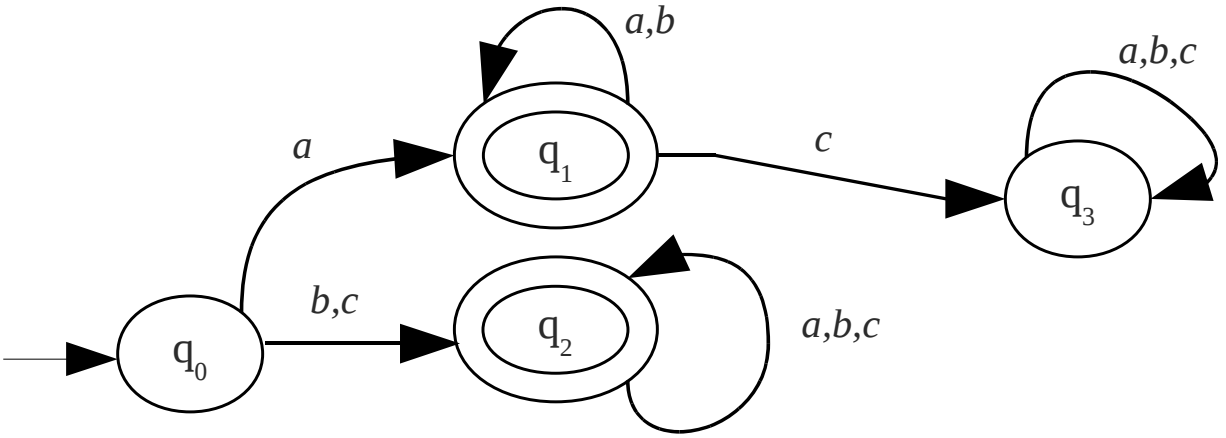


Figura 28.77: Autómata que reconoce el lenguaje $a(a|b)^*|(b|c)(a|b|c)^*$, p.e. la cadena abb pertenecería al lenguaje que reconoce el autómata, sin embargo la cadena acc no pertenecería a dicho lenguaje.

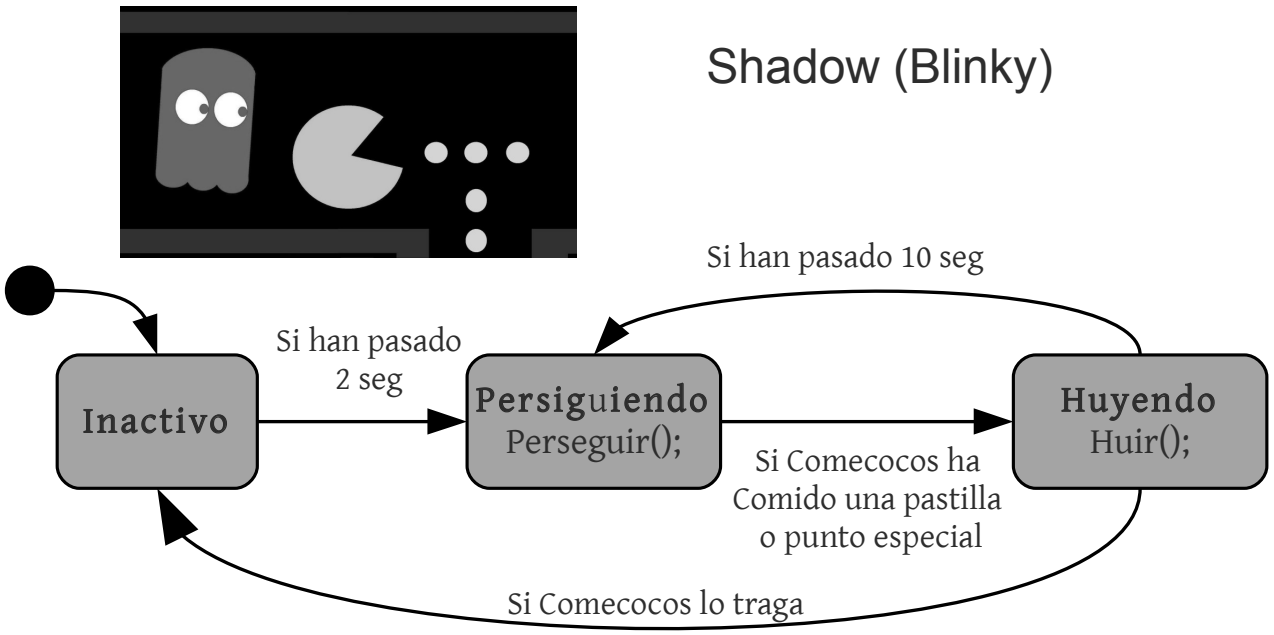


Figura 28.78: Comportamiento del fantasma Shadow (Blinky) del juego del Comecocos (PacMan), modelado por medio de una máquina de Moore.

Se emplearán las máquinas de Moore o Mealy para modelar los comportamientos en los videojuegos. Estas máquinas son autómatas de estados finitos determinista que poseen salida más allá de una decisión sobre si una cadena pertenece o no al lenguaje que reconoce el autómata.

Un ejemplo de uso de autómatas para modelar comportamientos en videojuegos es el PacMan o como se denominó en España, el "Comecocos". En este videojuego clásico creado por Toru Iwatani de la empresa Namco, se emplearon autómatas muy simples para modelar el comportamiento de los fantasmas. Los fantasmas presentaban dos comportamientos principales: "Persiguiendo" y "Huyendo", y un tercero menos importante "Inactivo". En el primero el fantasma perseguía al Comecocos para comerlo y en el segundo tenía que huir del Comecocos ya que este podía tragarlo. El estado "Inactivo" ocurría al comienzo del juego y después de haber sido tragado, donde los fantasmas se regeneran en una caja situada en el centro del laberinto durante un tiempo. Existían cuatro fantasmas (Shadow (Blinky), Speedy (Pinky), Bashful (Inky) y Pokey (Clyde)) de colores rojo, rosa, cyan y naranja. Cada uno de ellos tenía sus propias características en la forma de perseguir y huir, por ejemplo, Blinky era muy rápido y tenía la habilidad de encontrar al Comecocos en el escenario, sin embargo Inky era muy lento y muchas veces evitaba el encuentro con el Comecocos.

El autómata tendrá por lo tanto tres estados uno para regenerarse, otro para huir y el tercero para perseguir, el cambio del estado "Perseguiendo" a "Huyendo" se producirá cuando el Comecocos coma unos puntos especiales de tamaño mayor (o en inglés Power Pellets) y el contrario cuando pase un determinado tiempo en ese estado de huida. En la figura 28.78 se muestra la máquina de Moore para modelar el comportamiento del fantasma Shadow, las funcionalidades "Perseguir" y "Huir" serán propias de este fantasma.

Otros ejemplos de uso de los autómatas en videojuegos, son por ejemplo en los juegos FPS, tipo Quake, donde se emplean para modelar los comportamientos de los personajes virtuales que aparecen, con estados como por ejemplo "Huyendo", "Disparando" o "Peleando" en los que el personaje huirá de algo, disparará algún arma o peleará de algún modo contra alguien. Incluso en este tipo de videojuegos se pueden usar los autómatas para dotar de un comportamiento a las armas que se emplean.

En juegos deportivos se emplean para simular el comportamiento de los jugadores virtuales, con estados como "Disparando", "Driblando", "Corriendo", "Parando", ...También se usan los autómatas para modelar el comportamiento del equipo, "Atacando", "Defendiendo",...

En las próximas secciones se profundizará en la idea de uso de máquinas de estado finito en la modelización de comportamientos y se mostrará como se pueden realizar las implementaciones de este tipo de autómatas. Se estudiará como este tipo de autómatas simples se hacen más difíciles de utilizar cuando se ponen en juego muchos estados y se trabaja con requisitos de rehusabilidad. Se mostrará como solución una variante de estos para controlar los comportamientos: los autómatas de estados finitos jerárquicos.

Implementación de autómatas finitos deterministas

Antes de mostrar como se podría hacer la implementación de agentes para su uso en videojuegos, se va a presentar distintas formas de realizar la implementación de un autómata de estados finito. El objetivo de los ejemplos que se muestran a continuación es simplemente describir distintas formas de implementar autómatas, no se busca obtener las implementaciones de la mejor "calidad" posible.

La primera forma que se va a presentar es mediante el uso de funciones (se construye una función por cada estado del autómata) y por medio de estructuras de selección múltiple (como el switch en C) se establecen las transiciones que ocurren entre los estados. A continuación se muestra el código en lenguaje C de la implementación del autómata de la figura 28.77 realizado de esta forma.

La implementación de un autómata se puede hacer, de una manera más eficiente que la expuesta antes, empleando una matriz. La matriz tendrá las filas etiquetadas con los estados del autómata y las columnas con los símbolos del alfabeto de entrada. Cada intersección $[i, j]$ informa del estado al que pasará el autómata si estando en el estado i se lee el símbolo j.

Listado 28.23: Autómata de la figura 28.77 (sentencias switch).

```
#include <stdio.h>

int estado_q0(void) {
  char c;
  c=getchar();
  switch (c) {
  case 'a': return estado_q1();
    break;
  case 'b':
  case 'c': return estado_q2();
    break;
  default:  return 0;
  }
}

int estado_q1(void) {
  int c;
  c=getchar();
  while ((c=='a') || (c=='b')) c=getchar();
  switch (c) {
  case '\n': return 1;
    break;
  case 'c':   return estado_q3();
    break;
  default:    return 0;
  }
}

int estado_q2(void) {
  int c;
  c=getchar();
  while ((c=='a') || (c=='b') || (c=='c')) c=getchar();
  switch (c) {
  case '\n': return 1;
    break;
  default:    return 0;
  }
}

int estado_q3(void) {
  int c;
  c=getchar();
  while ((c=='a') || (c=='b') || (c=='c')) c=getchar();
  switch (c) {
  case '\n': return 0;
    break;
  default:    return 0;
  }
}

int main (int argc,char *argv[]) {
  if(estado_q0())
    printf("Es una cadena del Lenguaje\n");
  else
    printf("\nNo es una cadena del Lenguaje \n");
  return 1;
}
```

La forma más general de la matriz se podría definir de la siguiente forma:

$$estado_siguiente = matriz[estado_actual][entrada]$$

Esto no es más que una implementación de la tabla de transiciones del autómata. La matriz asociada al autómata de la figura 28.77, es la que se muestra en la figura 28.79. El código en lenguaje C, que corresponde a la implementación del autómata de la figura 28.77 de esta forma, es el que se muestra a continuación.

	a	b	c
$\rightarrow q_0$	q_1	q_2	q_2
$\bullet\ q_1$	q_1	q_1	q_3
$\bullet\ q_2$	q_2	q_2	q_2
q_3	q_3	q_3	q_3

Figura 28.79: Tabla de Transición para el autómata de la figura 28.77.

Listado 28.24: Autómata de la figura 28.77 (tabla transición).

```
1  #include <stdio.h>
2
3  /* Se define la Tabla de Transicion del automata */
4
5  #define N 4 //numero de filas de la tabla de transicion
6  #define M 3 //numero de columnas de la tabla de transicion
7
8  int TablaTrans[N][M] = { 1,2,2,
9                           1,1,3,
10                          2,2,2,
11                          3,3,3};
12
13 int AFD() {
14   int estado_actual, estado_siguiente;
15   int c;
16
17   estado_actual=0;
18   c=getchar();
19
20   while (c!='\n') {
21     switch (c) {
22     case 'a': estado_siguiente=TablaTrans[estado_actual][0];
23       break;
24     case 'b': estado_siguiente=TablaTrans[estado_actual][1];
25       break;
26     case 'c': estado_siguiente=TablaTrans[estado_actual][2];
27       break;
28     default:  return 0;
29     }
30     if (c!='\n') estado_actual=estado_siguiente;
31     c=getchar();
32   }
33   if ((estado_actual==1) || (estado_actual==2))  return 1;
34   else return 0;
35 }
36
37 int main (int argc,char *argv[]) {
38   if(AFD())
39     printf("Es una cadena del Lenguaje\n");
40   else
41     printf("\nNo es una cadena del Lenguaje \n");
42   return 1;
43 }
```

Otra posibilidad es emplear una sentencia switch para seleccionar el estado y sentencias if-else para determinar las transiciones a partir de cada uno de los estados. En el siguiente código en C++ se muestra como sería la implementación del autómata de la figura 28.77.

Listado 28.25: Autómata de la figura 28.77 (sentencias condicionales).

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  enum StateType {q0, q1, q2, q3, error};
```

```
 7
 8 int main() {
 9   string palabra;
10   int n, i, mensaje=0;
11   StateType CurrentState;
12
13   cout << "Introduzca la cadena a analizar: "; cin >> palabra;
14   n = palabra.length(); i = 0;
15   CurrentState = q0;
16   while (i<=n-1) {
17     switch(CurrentState) {
18
19     case q0:
20       if (palabra[i]=='a') CurrentState=q1;
21       else if (palabra[i]=='b' || palabra[i]=='c') CurrentState=q2;
22       else CurrentState=error;
23       break;
24
25     case q1:
26       if (palabra[i]=='a' || palabra[i]=='b') CurrentState=q1;
27       else if (palabra[i]=='c') CurrentState=q3;
28       else CurrentState=error;
29       break;
30
31     case q2:
32       if ((palabra[i]=='a') || (palabra[i]=='b') || (palabra[i]=='c
             '))
33     CurrentState=q2;
34       else CurrentState=error;
35       break;
36
37     case q3:
38       if ((palabra[i]=='a') || (palabra[i]=='b') || (palabra[i]=='c
             '))
39     CurrentState=q3;
40       else CurrentState=error;
41       break;
42
43     default:
44       if (!mensaje) {
45     cout << "Error alfabeto no valido" << endl;
46     cout << palabra[i-1] << endl;
47     mensaje=1;
48       }
49     }
50     i++;
51   }
52
53   if ((CurrentState==q1) || (CurrentState==q2))
54     cout << "La cadena " + palabra + " pertenece al lenguaje." <<
           endl;
55   else cout << "La cadena " + palabra + " NO pertenece al lenguaje.
         " << endl;
56
57   return 0;
58 }
```

Y para finalizar se mostrará como se puede realizar la implentación de un autómata finito determinista empleando la programación orientada a objetos. En el siguiente código en C++ se muestra como se podría realizar en dicho lenguaje.

Listado 28.26: Autómata de la figura 28.77 (objetos).

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
```

```
6  enum StateType {q0, q1, q2, q3, error};
7
8  string palabra;
9  int n, i, mensaje = 0;
10
11 class automata {
12   StateType CurrentState;
13
14 public:
15   void Inicializar(void);
16   void ChangeState(StateType State);
17   void UpdateState(void);
18   StateType Informar(void);
19 };
20
21 void automata::Inicializar(void) {
22   CurrentState=q0;
23 }
24
25 StateType automata::Informar(void) {
26   return CurrentState;
27 }
28
29 void automata::ChangeState(StateType State) {
30     CurrentState=State;
31 }
32
33 void automata::UpdateState(void) {
34   switch(CurrentState) {
35   case q0:
36     if (palabra[i]=='a') ChangeState(q1);
37     else if (palabra[i]=='b' || palabra[i]=='c') ChangeState(q2);
38     else ChangeState(error);
39     break;
40
41   case q1:
42     if (palabra[i]=='a' || palabra[i]=='b') ChangeState(q1);
43     else if (palabra[i]=='c') ChangeState(q3);
44     else ChangeState(error);
45     break;
46
47   case q2:
48     if ((palabra[i]=='a') || (palabra[i]=='b') || (palabra[i]=='c')
         )
49       ChangeState(q2);
50     else ChangeState(error);
51     break;
52
53   case q3:
54     if ((palabra[i]=='a') || (palabra[i]=='b') || (palabra[i]=='c')
         )
55       ChangeState(q3);
56     else ChangeState(error);
57     break;
58
59   default:
60     if (!mensaje) {
61       cout << "Error alfabeto no valido" << endl;
62       cout << palabra[i-1] << endl;
63       mensaje=1;}
64   }
65 }
66
67 int main() {
68   automata ejemplo;
69   StateType State;
70
71   cout<<"Introduzca la cadena a analizar: ";
72   cin>>palabra;
73   n = palabra.length();
```

```
74    i = 0;
75    ejemplo.Inicializar();
76
77    while (i<=n-1) {
78      ejemplo.UpdateState();
79      i++;
80    }
81
82    State=ejemplo.Informar();
83    if ((State==q1) || (State==q2))
84      cout << "La cadena " + palabra + " pertenece al lenguaje." <<
            endl;
85    else cout << "La cadena " + palabra + " NO pertenece al lenguaje.
          " << endl;
86    return 0;
87  }
```

Se ha creado una clase llamada "*automata*", que se usará para crear un objeto autómata. Esta clase contiene una variable privada, **CurrentState**, que almacenará el estado en el que se encuentra el autómata en cada momento. A esta variable solo se podrá acceder desde las funciones miembro de la clase, que son: **Inicializar()**, **ChangeState()**, **UpdateState()** y **Informar()**. La función **Inicializar(()** es una función empleada para poner al autómata en el estado inicial. La función **ChangeState()** es empleada para cambiar el estado en el que se encuentra el autómata.

La función **Informar()** devuelve el estado en el que está el autómata en un instante cualquiera. Por último, la función **UpdateState()** es la que implementa la tabla de transiciones del autómata. Para conseguir el funcionamiento deseado lo que se hará será crear un objeto de la clase **automata**, posteriormente será inicializado, luego se consultará la tabla de transiciones (ejecuciones de la función **UpdateState()**) mientras la cadena que se esté analizando tenga elementos, momento en el que se consultará el estado en el que se queda el autómata tras leer toda la cadena para chequear si es un estado de aceptación o no.

En todas las implementaciones, la cadena de entrada será aceptada si, una vez leída, el estado que se alcanza es uno perteneciente al conjunto de estados finales. En caso contrario, la cadena no será un elemento del lenguaje que reconoce el autómata.

28.5.4. Implementación de agentes basados en autómatas

En el campo del desarrollo de videojuegos, como ya ha sido comentado, se suele trabajar con unas máquinas especiales, las máquinas de Moore o Mealy, que extienden los autómatas de estados finitos (FSM) para dotarlos de mayor expresividad. En la figura 28.80 se muestra un autómata del tipo de los que se usarán en el diseño de videojuegos.

Los c_i que aparecen en el autómata mostrado son estados, cada uno de los cuales tiene asociado una funcionalidad, o en el caso de los videojuegos un comportamiento, diferente. El autómata que se muestra en la figura 28.80 tiene 4 estados y su funcionamiento contará por lo tanto con otros tantos comportamientos $\{c_0, c_1, c_2, c_3\}$. Las r_{ij} son reglas que son disparadas ante estímulos que ocurren en el entorno y que le permiten cambiar de estado y por lo tanto modificar su comportamiento. Por ejemplo, si estando en el estado c_0 se satisfacen unas condiciones en el entorno que permite el disparo de la regla r_{01}, esto provocará un cambio de estado, del estado c_0 al estado c_1 y por lo tanto un cambio de comportamiento. También se puede observar como desde el estado c_3 podrían dispararse dos reglas diferentes r_{31} y r_{32}, en estos casos los estímulos que disparan cada regla deben ser diferentes, se recuerda la necesidad de estar en un único estado en cada instante del funcionamiento del autómata (son deterministas).

Los juegos que son desarrollados usando lenguajes de alto nivel, como por ejemplo C o C++, típicamente almacenan todos los datos relacionados a cada agente en una estructura o clase. Esta estructura o clase puede contener variables para almacenar información como la posición, vida, fuerza, habilidades especiales, y armas, entre muchas otras. Por supuesto, junto a todos estos elementos, en la estructura o clase también se almacenará el estado actual del autómata, y será éste el que determine el comportamiento del agente.

En el siguiente código se muestra cómo se podría almacenar los datos que definen al agente dentro de una clase.

Listado 28.27: Clase contenedora de los datos de un agente.

```
1  class AIEntity
2  {
3   public:
4    int type;
5    int state;
6    int row;
7    int column;
8    int health;
9    int strength;
10   int intelligence;
11   int magic;
12 };
```

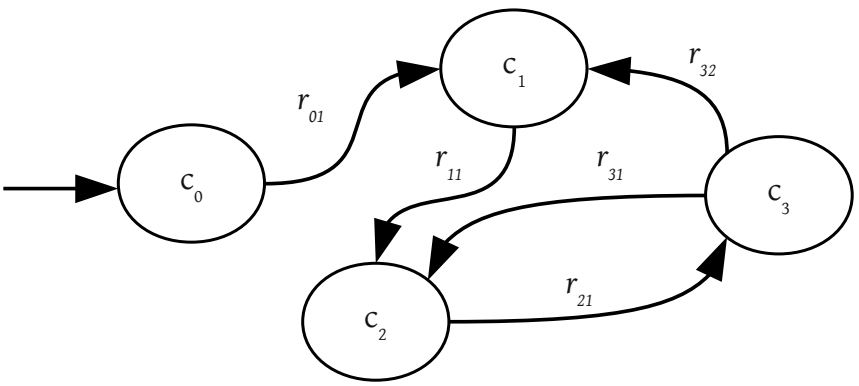


Figura 28.80: Forma de los Autómatas con los que se va a trabajar

Para implementar este tipo de máquinas de estado finito dotadas con salida, o más concretamente agentes como los comentados en la sección anterior para su uso en videojuegos, existen varias alternativas. Éstas están basadas en las que ya se han presentado en la sección anterior pero con mejoras. La manera más simple es por medio de múltiples sentencias "If-Else" o la mejora inmediata al uso de estas sentencias, por medio de una sentencia "switch". La sentencia **switch(estado_actual)** es una sentencia de selección. Esta sentencia permite seleccionar las acciones a realizar de acuerdo al valor que tome la variable "estado_actual", habrá un "case" para cada estado posible. La forma general de usar esta sentencia para implementar una máquina de estados es la siguiente:

Listado 28.28: Implementación del autómata por medio de sentencias switch.

```
switch (estado_actual)
{
    case estado 1:
        Funcionalidad del comportamiento del estado 1;
        Transiciones desde este estado;
        break;
    case estado 2:
        Funcionalidad del comportamiento del estado 2;
        Transiciones desde este estado;
        break;
    case estado N:
        Funcionalidad del comportamiento del estado N;
        Transiciones desde este estado;
        break;
    default:
        Funcionalidad por defecto;
}
```

A modo de ejemplo, se muestra como sería la implementación en C++ del autómata que simula el comportamiento del fantasma Shadow del videojuego del ComeCocos (vea la figura 28.78), partiendo de la implementación más avanzada que se presentó en la sección anterior, es decir con programación orientada a objetos y por medio de una sentencia "switch" y sentencias "if-else". No se entra en detalles sobre como se implementarían las funcionalidades asociadas a cada estado, aunque se recuerda que esas serían específicas de cada fantasma. Tampoco se muestra como sería la implementación de la clase "Ghost".

Listado 28.29: Implementación del autómata que simula el comportamiento del fantasma en el juego del ComeCocos.

```
enum StateType{Inactivo, Huyendo, Persiguiento};

void Ghost::UpdateState(StateType CurrentState, GhostType GType)
{

  switch(CurrentState)
    {

       case Huyendo:
           Huir(GType);
           if (PacManTraga())
             {
               ChangeState(Inactivo);
             }
            else if (Temporizador(10)) ChangeState(Persiguiendo);
           break;

       case Persiguiendo:
           Perseguir(GType);
           if (PacManComePastilla()) ChangeState(Huyendo);
           break;

        case Inactivo:
           Regenerar();
           if (Temporizador(2)) ChangeState(Persiguiendo);
           break;
    }
}
```

Figura 28.81: Los juegos de francotiradores son juegos con gran aceptación (p.e. Sniper Elite V2), probablemente por estar conectados con una niñez en la que juegos como el escondite o el pilla-pilla eran toda una aventura.

Otro ejemplo más sobre esta forma de implementar los autómatas, se muestra en el siguiente código, que corresponde a la implementación de la máquina de estados finitos que podría simular el comportamiento de un soldado que se enfrenta a un francotirador (el jugador) en un juego FPS, tipo Sniper Elite V2 (vea la figura 28.81).

Como se puede observar en el código existen cinco estados: *Huyendo*, *Patrullando*, *Persiguiendo*, *Disparando* e *Inactivo*. El estado inicial podría ser Patrullando. En este estado el soldado estará observando el entorno buscando al francotirador (el jugador). Si lo encuentra lo perseguirá (pasando al estado Persiguiendo). Si por contra recibe un ataque huirá (pasando al estado Huyendo). En el estado Persiguiendo intentará poner a tiro al francotirador, si lo consigue le disparará, pasando al estado Disparando. Pero si en la persecución, es el francotirador el que le amenaza por tener una mejor posición, provocará que el soldado pase al estado Huyendo para evitar el ataque. En el estado Disparando, el soldado dispara si cree que puede abatir al francotirador, si no, lo persigue hasta volver a conseguirlo. En el estado Huyendo un soldado huirá hasta que logre ponerse a salvo, momento en el que pasa al estado Patrullando. Si no logra ponerse a salvo es porque haya sido cazado por el francotirador por lo que pasará a un estado de inactividad (estado Inactivo). En este estado el soldado se regenerará y pasado un tiempo se posicionará sobre el escenario, donde volverá a patrullar en el estado Patrullando.

Listado 28.30: Implementación del autómata que simula el comportamiento de un soldado que se enfrenta a un francotirador en un juego de tipo FPS (p.e. Sniper Elite V2)

```
enum StateType{Patrullando, Persiguiendo, Disparando, Huyendo,
     Inactivo};

void Soldier::UpdateState(StateType CurrentState)
{
  switch(CurrentState)
   {
     case Patrullando:
     Patrullar();
     if (ObjetivoLocalizado()) ChangeState(Persiguiendo);
           else (AtaqueRecibido()) ChangeState(Huyendo);
     break;

     case Persiguiendo:
         Perseguir();
         if (ObjetivoVulnerable()) ChangeState(Disparando);
              else ChangeState(Huyendo);
         break;

     case Disparando:
         if (ObjetivoAbatible()) Disparar();
           else ChangeState(Persiguiendo);
         break;

     case Huyendo:
         Huir();
         if (PuestoaSalvo()) ChangeState(Patrullando);
           else if (Abatido()) ChangeState(Inactivo);
         break;

     case Inactivo:
         Regenerar();
         if (TranscurridoTiempo()) ChangeState(Patrullando);
         break;
   }
}
```

Aunque a primera vista, este enfoque puede parecer razonable por lo menos en autómatas no complejos, cuando se aplica a juegos, o mejor aún, a comportamientos de objetos más complejos, la solución presentada se convierte en un largo y tortuoso camino. Cuantos más estados y condiciones se tengan que manejar, más riesgo hay de convertir este tipo de estructura en código spaghetti (vea la figura 28.82), por lo que el flujo del programa será difícil de entender y la tarea de depuración una pesadilla. Además, este tipo de implementación es difícil de extender más allá del propósito de

Estado	Condición	Estado_Destino
Inactivo	Regenerado	Persiguiendo
Persiguiendo	PastillaComida	Huyendo
Huyendo	TranscurridoTiempo	Persiguiendo
Huyendo	Comido	Inactivo

Cuadro 28.6: Tabla de Transición para el autómata implementado para simular el comportamiento del fantasma en el juego del Comecocos (vea Figura 28.78.

su diseño original. Esta flexiblidad en el diseño es una característica deseable ya que las extensiones son a menudo muy frecuentes en este campo. Con frecuencia habrá que ajustar el comportamiento del agente (por ende el autómata) para hacer frente a las circunstancias no planificadas y conseguir un comportamiento como el que se esperaba.

Figura 28.82: Código "spaghetti"'. Término peyorativo para los programas de computación que tienen una estructura de control de flujo compleja e incomprensible.

Otro aspecto que hace poco interesante este tipo de implementación es la asociación de eventos con los estados, es decir puede que interese hacer que ocurra algo cuando se entre o salga de un estado, Por ejemplo, en el juego del Comecocos, cuando el Comecocos come alguna de las pastillas especiales, se puede querer (de hecho es así) que los fantasmas entren en el estado de huída y cambien su color a azul. La inclusión de este tipo de eventos asociados a la entrada y salida en esta implementación por medio de sentencias switch/if-else implican que ocurran en el cambio de estado más que en el estado propiamente dicho, y complicarían aún más el código.

Implementación basada en tabla de transición

El uso de la tabla de transición con alguna modificación podría también ser útil en la implementación de los agentes. La forma de implementar un autómata finito determinista empleando su tabla de transición ya ha sido presentada en la sección anterior. Esta tabla almacenará los cambios de estado que ocurren cuando estando en un determinado estado se satisface alguna condición. La Tabla 28.6 muestra un ejemplo de como sería esta tabla para modelar el comportamiento del fantasma en el juego del Comecocos.

A continuación se muestra como sería la implementación del autómata que modela el comportamiento del fantasma en el juego del Comecocos por medio de la tabla de transiciones (vea la Tabla 28.6 y figura 28.78). Se han añadido las funcionalidades del fantasma y un mecanismo para la recepción de estímulos de una forma muy simplificada, con el objetivo de permitir la comprobación de su funcionamiento.

La tabla será consultada por el agente cada vez que reciba estímulos, éstos serán chequeados para ver si le permite realizar alguna transición con el objeto de cambiar de estado. Cada estado podría ser modelado como un objeto independiente del agente, ofreciendo de esta forma una arquitectura limpia y flexible.

Listado 28.31: Implementación del autómata que simula el comportamiento del fantasma en el juego del Comecocos

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 #define N 4 //numero de filas de la tabla de transicion
7 #define M 4 //numero de columnas de la tabla de transicion
```

```
 8
 9 enum StateType {inactivo, persiguiendo, huyendo, error};
10 enum Rule {regenerado, pastillacomida, transcurridotiempo, comido};
11
12 class ghost {
13
14   StateType CurrentState;
15
16   /* Se define la Tabla de Transicion del automata */
17   StateType TablaTrans[N][M];
18
19 public:
20   void Inicializar(void);
21   void ChangeState(StateType State);
22   void UpdateState(Rule Transicion);
23   StateType Informar(void);
24   void Huir(void);
25   void Perseguir(void);
26   void Regenerar(void);
27   void Error(void);
28 };
29
30 void ghost::Inicializar(void) {
31   CurrentState=inactivo;
32   TablaTrans[0][0] = persiguiendo; TablaTrans[0][1] = error;
33   TablaTrans[0][2] = error;        TablaTrans[0][3] = error;
34   TablaTrans[1][0] = error;        TablaTrans[1][1] = huyendo;
35   TablaTrans[1][2] = error;        TablaTrans[1][3] = error;
36   TablaTrans[2][0] = error;        TablaTrans[2][1] = error;
37   TablaTrans[2][2] = persiguiendo; TablaTrans[2][3] = inactivo;
38   TablaTrans[3][0] = error;        TablaTrans[3][1] = error;
39   TablaTrans[3][2] = error;        TablaTrans[3][3] = error;
40 }
41
42 StateType ghost::Informar(void) {
43   return CurrentState;
44 }
45
46 void ghost::ChangeState(StateType State) {
47     CurrentState=State;
48 }
49
50 void ghost::UpdateState(Rule Transicion) {
51   StateType NextState;
52
53   NextState=TablaTrans[CurrentState][Transicion];
54   ChangeState(NextState);
55
56   switch(CurrentState) {
57   case huyendo:
58     Huir();
59     break;
60   case persiguiendo:
61     Perseguir();
62     break;
63   case inactivo:
64     Regenerar();
65     break;
66   default: Error();
67   }
68 }
69
70 /* Funciones de prueba */
71 void ghost::Huir(void) {
72   cout << "El fantasma huye!" << endl;
73 }
74
75 void ghost::Perseguir(void) {
76   cout << "El fantasma te persigue!" << endl;
77 }
78
```

```
79 void ghost::Regenerar(void) {
80   cout << "El fantasma se regenera!" << endl;
81 }
82
83 void ghost::Error(void) {
84   cout << "El fantasma esta en un estado de error!" << endl;
85 }
86
87 int main() {
88   ghost shadow;
89   StateType State;
90   string palabra;
91   shadow.Inicializar();
92   cout<<"Introduzca la regla: ";
93   cin>>palabra;
94
95   while (palabra.compare("salir")!=0) {
96     if (palabra.compare("regenerado")==0)
97       shadow.UpdateState(regenerado);
98     else if (palabra.compare("pastillacomida")==0)
99       shadow.UpdateState(pastillacomida);
100    else if (palabra.compare("transcurridotiempo")==0)
101      shadow.UpdateState(transcurridotiempo);
102    else if (palabra.compare("comido")==0)
103      shadow.UpdateState(comido);
104    else cout << "Accion no valida" << endl;
105    cout<<"Introduzca la regla: ";
106    cin>>palabra;
107  }
108
109  State=shadow.Informar();
110
111  if (State==error)
112    cout << "El fantasma no funciona bien." << endl;
113  else cout << "El fantasma funciona perfecto." << endl;
114
115  return 0;
116 }
```

Implementación basada en patrones de diseño

Hasta ahora, la mayoría de las implementaciones de autómatas que se han presentado hacen uso de la sentencia switch, que cuando son usadas para modelar comportamientos no muy complejos parece más que adecuado. Sin embargo esta sentencia, que permite una programación tipo goto, se complica cuando los comportamientos son complejos. En estos casos se requerirán muchas ramas, una por estado, y cada una de ellas tendrá largas sentencias condicionales para establecer las transisiones. Cada vez que deba ser cambiada la sentencia switch, porque sea necesario añadir o eliminar algún estado, por lo general acarreará un montón de cambios en el resto de la implementación. A continuación se muestra una posible solución, basada en objetos, para la implementación de autómatas, y por lo tanto de agentes, que evita la sentencia switch.

Patrón de diseño

Un patrón de diseño es una solucion simple y elegante a un problema específico y común en el diseño orientado a objetos, que está basada en la experiencia y su funcionamiento ha sido demostrado. Su objetivo es permitir la construcción de software orientado a objetos flexible y fácil de mantener.

El fin que se prentende lograr con la sentencia switch es controlar el comportamiento en función del estado. En el mundo orientado a objetos un cambio de comportamiento implica un tipo diferente de objeto. Por lo tanto, en este caso, lo que se necesita es un objeto que parezca cambiar su tipo en función de su estado interno. Lo que se va a hacer es intentar describir el comportamiento de un objeto como una colección de estados, de manera que sea el propio objeto el que maneje su comportamiento mediante transiciones de un estado a otro. Para ello se va a hacer uso del patrón de diseño *State*, cuyo diagrama UML es mostrado en la figura 28.83. El patrón de diseño *state* es una solución al problema de cómo hacer que un comportamiento dependa del estado.

Las clases y/o objetos participantes en este patrón son:

- Context Class, esta clase define la interfaz, y es la responsable de mantener el estado actual. Esta será la clase que use la aplicación.
- State Class, esta clase define la interfaz para encapsular el comportamiento asociado con un estado particular de la clase Context.
- ConcreteState Class, cada una de las subclases que implementa un comportamiento asociado con un estado del contexto.

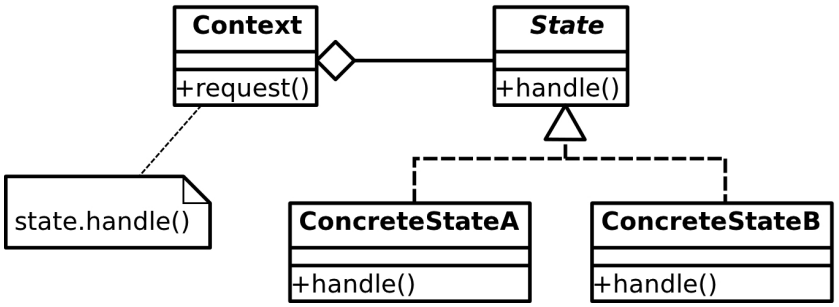


Figura 28.83: Diagrama UML del Patrón de diseño State.

La idea de este patrón es tener una clase, Contexto (Context class), que tendrá un comportamiento dependiendo del estado actual en el que se encuentre, es decir un comportamiento diferente según su estado. También tiene una clase Estado abstracta (State Class) que define la interfaz pública de los estados. En ella, se pondrán todas las funciones del Contexto cuyo comportamiento puede variar. Luego hay una implementación de los estados concretos, que heredarán de la clase Estado abstracto. La clase Contexto, poseerá un puntero sobre el estado actual, que será almacenado en una variable miembro de la clase. Cuando se desea que el Contexto cambie de estado, solo hay que modificar el estado actual.

El uso de este patrón implicará:

- Definir una clase “contexto” para presentar una interfaz simple al exterior.
- Definir una clase base abstracta “estado”.
- Representar los diferentes “estados” del autómata como clases derivadas de la clase base “estado”.
- Definir la conducta específica de cada estado en la clase apropiada de las definidas en el paso anterior (derivadas de la clase abstracta “estado”).
- Mantener un puntero al estado actual en la clase “contexto”.
- Para cambiar el estado en el que se encuentra el autómata, hay que cambiar el puntero hacia el estado actual.

La implementación del autómata que simula el comportamiento del fantasma en el juego del Comecocos empleando el patrón de diseño *State* se muestra en el siguiente listado de código.

El patrón de diseño State no especifica donde son definidas las transiciones entre estados. Esto se puede hacer de dos formas diferentes: en la clase “Context”, o en cada estado individual derivado de la clase “State”. En el código mostrado anteriormente se ha realizado de esta segunda forma. La ventaja de hacerlo así es la facilidad que ofrece para añadir nuevas clases derivadas de la clase “State”. Su inconveniente es que cada estado derivado de la clase “State” tiene que conocer como se conecta con otras clases, lo cual introduce dependencias entre las subclases.

A modo de resumen, conviene indicar que no hay una única manera de decribir estados. Dependiendo de la situación podría ser suficiente con crear una clase abstracta llamada State que posea unas funciones Enter(), Exit() y Update(). La creación de los estados particulares del autómata que se quiera implementar implicaría crear una subclase de la clase State por cada uno de ellos.

Listado 28.32: Implementación del autómata que simula el comportamiento del fantasma en el juego del Comecocos.

```
#include <iostream>
#include <string>

using namespace std;

class FSM;

// Clase base State.
// Implementa el comportamiento por defecto de todos sus métodos.
class FSMstate {
public:
  virtual void regenerado(  FSM* )  {
    cout << "    No definido" << endl; }
  virtual void pastillacomida(  FSM* )  {
    cout << "    No definido" << endl; }
  virtual void transcurridotiempo(  FSM* )  {
    cout << "    No definido" << endl; }
  virtual void comido(  FSM* )  {
    cout << "    No definido" << endl; }
protected:
  void changeState(FSM*, FSMstate*);
};

// Automa context class. Reproduce el intefaz de la clase State y
// delega todo su comportamiento a las clases derivadas.
class FSM {
public:
  FSM();
  void regenerado()          { _state->regenerado(this); }
  void pastillacomida()      { _state->pastillacomida(this); }
  void transcurridotiempo() { _state->transcurridotiempo(this); }
  void comido()              { _state->comido( this ); }

private:
  friend class FSMstate;
  void changeState( FSMstate* s )  { _state = s; }
  FSMstate*  _state;
};

void FSMstate::changeState( FSM* fsm, FSMstate* s ) {
  fsm->changeState( s ); }

// Clase derivad de State.
class Inactivo : public FSMstate {
public:
  static FSMstate* instance() {
    if ( ! _instance ) _instance = new Inactivo;
    cout << "El fantasma se regenera!" << endl;
    return _instance; };
  virtual void regenerado( FSM* );
```

```
51 private:
52   static FSMstate* _instance; };
53
54 FSMstate* Inactivo::_instance = 0;
55
56 class Persiguiendo : public FSMstate {
57 public:
58   static FSMstate* instance() {
59     if ( ! _instance ) _instance = new Persiguiendo;
60     cout << "El fantasma te persigue!" << endl;
61     return _instance; };
62   virtual void pastillacomida( FSM* );
63 private:
64   static FSMstate* _instance; };
65
66 FSMstate* Persiguiendo::_instance = 0;
67
68 class Huyendo : public FSMstate {
69 public:
70   static FSMstate* instance() {
71     if ( ! _instance ) _instance = new Huyendo;
72     cout << "El fantasma huye!" << endl;
73     return _instance; };
74   virtual void transcurridotiempo( FSM* );
75   virtual void comido( FSM* );
76
77 private:
78   static FSMstate* _instance; };
79 FSMstate* Huyendo::_instance = 0;
80
81 void Inactivo::regenerado( FSM* fsm ) {
82   cout << "Cambio de Estado a Persiguiendo." << endl;
83   changeState( fsm, Persiguiendo::instance()); };
84
85 void Persiguiendo::pastillacomida( FSM* fsm ) {
86   cout << "Cambio de Estado a Huyendo." << endl;
87   changeState( fsm, Huyendo::instance()); };
88
89 void Huyendo::transcurridotiempo( FSM* fsm ) {
90   cout << "Cambio de Estado a Persiguiendo." << endl;
91   changeState( fsm, Persiguiendo::instance()); };
92
93 void Huyendo::comido( FSM* fsm ) {
94   cout << "Cambio de Estado a Inactivo." << endl;
95   changeState( fsm, Inactivo::instance()); };
96
97 // El estado de comienzo es Inactivo
98 FSM::FSM() {
99   cout << "Inicializa:" << endl;
100  changeState( Inactivo::instance() );
101 }
102
103 int main() {
104   FSM   ghost;
105   string  input;
106
107   cout<<"Introduzca regla: ";
108   cin>> input;
109
110   while (input.compare("salir")!=0) {
111     if (input.compare("regenerado")==0)
112       ghost.regenerado();
113     else if (input.compare("pastillacomida")==0)
114       ghost.pastillacomida();
115     else if (input.compare("transcurridotiempo")==0)
116       ghost.transcurridotiempo();
117     else if (input.compare("comido")==0)
118       ghost.comido();
119     else cout << "Regla no existe, intente otra" << endl;
120
121     cout<<"Introduzca regla: ";
```

```
122      cin>> input;
123    }
124  }
```

Pros y Contras de cada tipo de implementación

A continuación se determinan los pros y los contras principales de cada uno de estos dos métodos usados para implementar un atómata:

- **Usando sentencias Switch-If/Else**. Como pros hay que destacar dos, la primera es que es muy fácil añadir y chequear condiciones que permitan cambiar de estado, y la segunda es que permite la construcción rápida de prototipos. La principal contra es que cuando se tiene un número elevado de estados, el código se puede hacer con relativa rapidez, muy enredado y díficil de seguir (código "spaghetti"'). Otra contra importante, es que es difícil programar efectos asociados a la entrada salida de un estado (y se recuerda que esto es muy habitual en el desarrollo de videojuegos).
- **Usando una implementación orientada a objetos**. Este método posee como Pros:
  - *Altamente extensible*: Incluir un nuevo estado simplemente implica crear una nueva clase que hereda de la clase abstracta State.
  - *Fácil de mantener*: Cada estado reside en su propia clase, se podrá ver fácilmente las condiciones asociadas a ese estado que le permiten cambiar de estado sin tener que preocuparse acerca de los otros estados.
  - *Intuitivo*: Este tipo de implementación es más fácil de entender cuando se usa para modelar comportamientos complejos.

  Como contra principal, hay que destacar el tiempo de aprendizaje de uso de este método que es superior al anterior.

Avances en el uso de autómatas

Como ya se ha mostrado a lo largo de esta sección, los autómatas finitos son de gran utilidad a la hora de controlar el comportamiento de los agentes inteligentes, sin embargo, conforme crece el número de estados y se trabaja con requisitos de reusabilidad este tipo de autómatas se hacen más difíciles de utilizar. Una solución a este problema son los autómatas finitos jerárquicos HFSM (Hierarchical Finite State Machine). Estos no son más que autómatas finitos en los que cada estado puede contener otra máquina de estado subordinada. Los estados contenedores se denominan "superestados". El uso de superestados permite la agrupación de comportamientos, facilitando de esta forma la reusabilidad y la tarea de ampliar detalles en los estados.

En esta jerarquía de máquinas de estado, las máquinas de niveles superiores proporcionan el comportamiento general y el marco para ensamblar comportamientos más específicos, modelados por medio de máquinas de estado de niveles inferiores que son anidadas en estados del autómata de nivel superior. El uso de HFSM no reduce el número de estados, más bien al contrario, los aumenta al introducir los superestados. Donde si hay una reducción, y además considerable, es en el número de transiciones. Esta reducción es lo que hace este tipo de autómatas más simples, facilitando de este modo su representación y análisis.

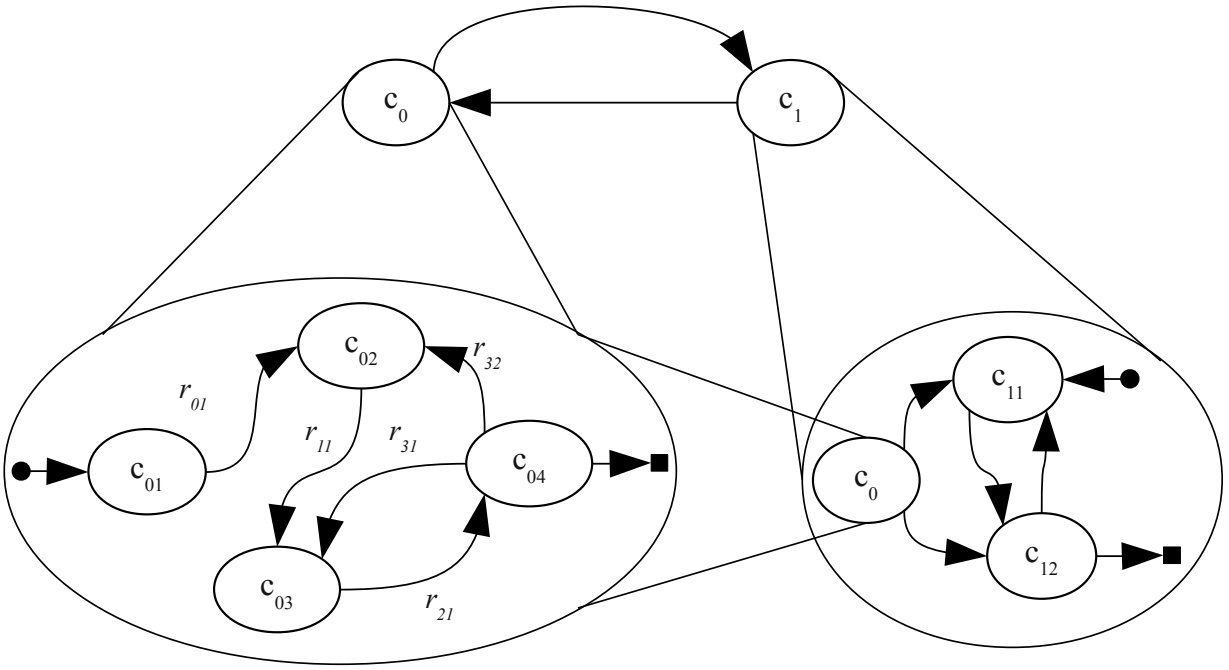


Figura 28.84: Forma de los Autómatas finitos jerárquicos.

A nivel de funcionamiento las HFSMs no aportan nada al modelo FSM. Existe un conjunto de estados y transiciones que llevan de un estado a otro. Alcanzar un superestado implica emplear el autómata contenido en él. Este autómata tendrá un punto de entrada y uno de salida, el alcanzar un superestado supone la ejecución de su autómata asociado desde su punto de entrada, por contra alcanzar el punto de salida implicará subir un nivel en la jerarquía.

Una transición que quede activa de un superestado a otro interrumpe el proceso que esté actualmente en marcha en grados jerárquicos menores. Respecto al resto de elementos, el funcionamiento es exactamente el mismo a las máquinas de estados finitos definidas anteriormente.

El uso de HFSM para el modelado de comportamientos a la hora de desarrollar agentes permite:

- **Aumentar el nivel de abstracción:** La jerarquía establece distintos niveles de detalle para cada una de las máquinas de estado. La consideración del superestado como una caja negra, permite la ocultación de los detalles presentes en los niveles más bajos de la jerarquía.
- **Facilitar la tarea de desarrollo:** La jerarquía permite seguir un diseño top-down. Se divide el problema en componentes, empezando por los niveles más altos, definiendo superestados abstractos, para posteriormente trabajar dentro de cada uno de esos estados abstractos de la misma manera.
- **Aumentar la modularidad:** Cada máquina de estado de la jerarquía puede ser vista y considerada como un módulo, esto facilita su reuso en otros puntos del desarrollo.

La figura 28.84 muestra un autómata finito jerárquico.

28.5.5. Usando los objetivos como guía de diseño

La manera más usual de implementar agentes, para su uso en videojuegos es, como ya se ha comentado, utilizando aproximaciones reactivas. En estas aproximaciones los agentes reaccionan ante las situaciones del juego mediante una acción o una secuencia de acciones predefinidas. Esto obliga a que los diseñadores de los comportamientos tengan que pensar a priori en todas las posibilidades que pueden darse y cómo deben reaccionar en ellas los agentes. Esto es un problema ya que provoca un comportamiento repetitivo y por lo tanto predecible. Es por ello por lo que en la actualidad se está tendiendo a usar otro tipo de agentes.

En la sección 28.5.2 se presentaron otras posibilidades para construir agentes, una de ellas eran los agentes deliberativos. Estos agentes poseen su propia representación interna del entorno y tienen capacidad para planificar una secuencia de acciones y ejecutarlas para alcanzar el fin para el que han sido diseñados.

La planificación es un área clave en la Inteligencia Artificial. Tradicionalmente se ha tratado como un problema de búsqueda de un plan que permitiera cambiar el estado actual del entorno al estado deseado. Por ejemplo, el planificador STRIPS (Stanford Research Institute Problem Solver), dado un estado inicial, trataba de encontrar un plan para que se cumpliera un conjunto de objetivos.

Un plan puede ser considerado como una secuencia de operadores, cada uno de los cuales constará de:

- Una lista de precondiciones. Estas deben ser ciertas para poder aplicar el operador.
- Una lista de proposiciones a añadir. La aplicación del operador implica añadir estas proposiciones al estado actual.
- Una lista de proposiciones a eliminar. La aplicación del operador implica eliminar estas proposiones del estado actual.

Normalmente, estos sistemas trabajan en entornos estáticos, es decir el estado del entorno no cambia entre la planificación y la ejecución del plan, lo que supone una limitación en el campo del desarrollo de videojuegos. Un cambio en el estado en el que se hace el plan implica realizar una replanificación, es decir hacer un nuevo plan, partiendo del estado actual. Por otra parte, y como ya se indicó en la sección 28.5.2 el principal problema que presentaban los agentes deliberativos es que necesitaban un tiempo elevado de respuesta, lo cual los hacía poco útiles en el campo del desarrollo de videojuegos. Es por ello, por lo que se han tenido que desarrollar versiones simplificadas de planificadores que obtengan respuetas en tiempo real.

Planificación de acciones orientada a objetivos

La modelización de comportamientos basados en aproximaciones reactivas pueden provocar que los personajes del juego muestren comportamientos repetitivos y por ello predecibles. Las aproximaciones deliberativas otorgan mayor libertad a estos personajes, ya que les premiten adaptar sus acciones para que se ajusten a la situación en la que se encuentran.

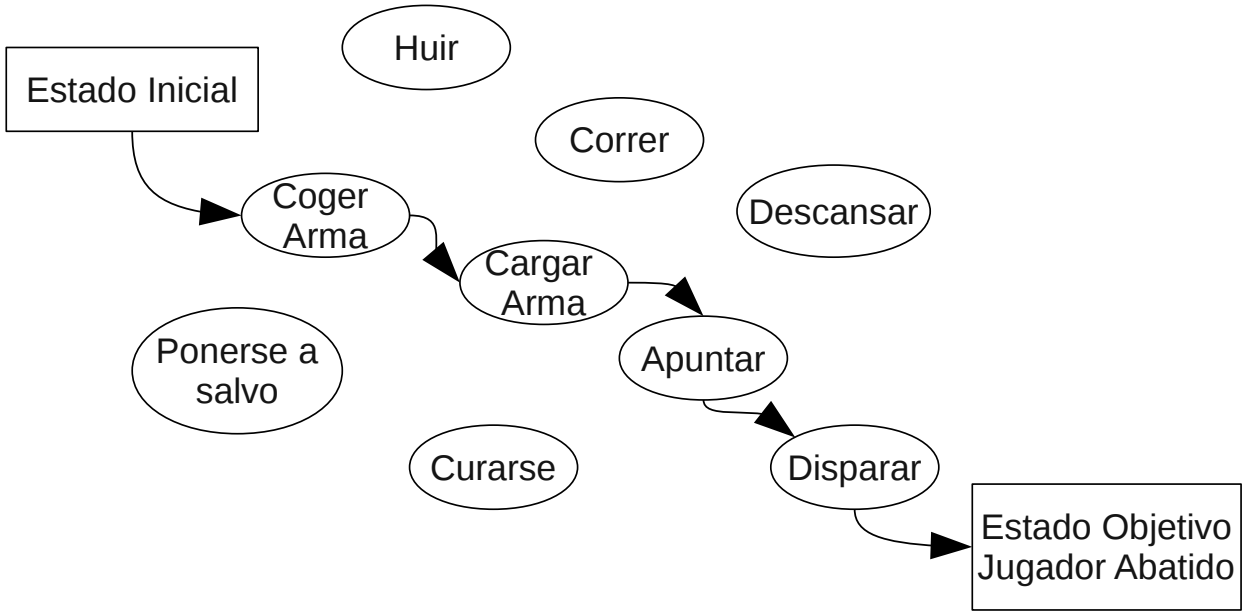


Figura 28.85: Formulación de un Plan.

La planificación de acciones orientada a objetivos, o GOAP (Goal-Oriented Action Planning), permite en tiempo real planificar el comportamiento de personajes en los juegos. Es una técnica de planificación que permite a los personajes decidir, no sólo qué hacer, sino también cómo hacerlo en tiempo real. Encontrarán soluciones alternativas a situaciones que ocurran durante el juego y gestionarán las situaciones no previstas durante el diseño (conviene notar que esto no se permite con las aproximaciones reactivas).

En el diseño basado en objetivos, los agentes son desarrollados sobre arquitecturas deliberativas. Un objetivo es una condición que un agente desea satisfacer. Los agentes pueden ser diseñados con uno o varios objetivos, pero en un instante concreto de tiempo, sólo uno podrá estar activo, y este será el que controle su comportamiento. Las acciones son las que van a permitir al agente satisfacer sus objetivos, y serán organizadas por medio de planes. Cada acción que pueda realizar el agente tendrá asociada unas condiciones que determinan cuando será posible su ejecución (es la precondición) y los efectos que tendrá en el estado del entorno su ejecución. Gracias a esto será posible el secuenciamiento de las acciones y su organización en planes.

El funcionamiento de este tipo de agentes es el siguiente: El sistema de generación de planes o planificador del agente buscará la secuencia de acciones (una acción es la aplicación un operador con unas precondiciones y unos efectos) que permita ir desde el estado inicial hasta el estado objetivo. Por lo general, el proceso de planificación implica buscar operadores cuyos efectos hagan cumplir los objetivos y haciendo de las precondiciones de estos operadores nuevos subobjetivos (encadenamiento hacia atrás).

El plan se ejecutará hasta que se complete, se invalide o hasta que haya un cambio de objetivo. Si cambia el objetivo o no se puede continuar con el plan actual por alguna razón, se aborta el plan actual y se genera uno nuevo. Existe también la posibilidad de que cuando exista más de un plan válido, el planificador obtenga el de menor coste, considerando que cada acción acarrea un coste. En cierto modo, tras todo este proceso lo que hay es una búsqueda de un camino o del camino de menor coste.

En la figura 28.85 se muestra un plan para matar al jugador, en el se pasa del estado inicial al estado objetivo mediante las acciones: Coger Arma, Cargar Arma, Apuntar y Disparar.

Con el uso de GOAP:

- Se consigue dotar a los personajes de comportamientos adaptativos. La capacidad de generación de planes permite ajustar sus acciones a su entorno actual, y encontrar diferentes soluciones a un problema.
- Se consigue comportamientos más simples de diseñar. Es el planificador quien se encarga de analizar las posibles alternativas existentes y encontrar la mejor opción, no hay que prever para cada situación posible la acción a realizar.
- Se consigue comportamientos variables. Los planes no están definidos de antemano, se elaboran en tiempo de ejecución dependiendo del estado actual del entorno.

Figura 28.86: Los desarrolladores del juego S.T.A.L.K.E.R.: Shadow of Chernobyl han usado agentes reactivos basados en FSM, después han empleado agentes basados en HFSM, para acabar usando agentes deliberativos basados en planificadores GOAP jerárquicos.

Ejemplos de juegos que emplean GOAP son:

- Mushroom Men: The Spore Wars (Wii) - Red Fly Studio, 2008
- Los Cazafantasmas (Wii) - Red Fly Studio, 2008
- Silent Hill: Homecoming (X360/PS3) - Double Helix Games / Konami de 2008
- Fallout 3 (X360/PS3/PC) - Bethesda Softworks, 2008
- Empire: Total War (PC) - Creative Assembly / Sega, 2009
- FEAR 2: Project Origin (X360/PS3/PC) - Monolith Productions Bros / Warner, 2009
- Demigod (PC) - Gas Powered Games / Stardock, 2009
- Just Cause 2 (PC/X360/PS3) - Avalanche Studios / Eidos Interactive, 2010
- Transformers: War for Cybertron (PC/X360/PS3) - High Moon Studios / Activision, 2010
- Trapped Dead (PC) - Juegos Headup, 2011
- Deus Ex: Human Revolution (PC/X360/PS3) - Eidos Interactive, 2011

La tendencia actual en el desarrollo de videojuegos es usar **planificadores jerárquicos**. Un planificador jerárquico primero formula un plan abstracto expresado en términos de tareas de alto nivel, por lo que no puede realizarse directamente. Después refina dicho plan con el objetivo de producir un plan completo en téminos de operadores básicos, o acciones que el agente pueda realizar directamente.

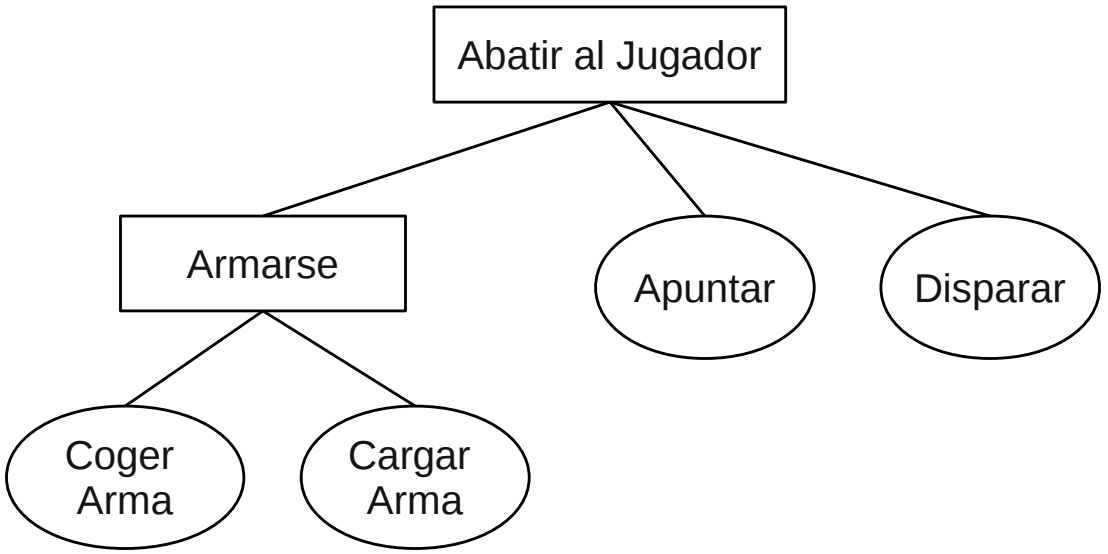


Figura 28.87: Planificador jerárquico.

En la figura 28.87 se muestra un ejemplo de uso de planificador jerárquico, en el se indica que para pasar de un estado inicial a un estado en el que el jugador haya muerto, se puede trazar un plan compuesto por una tarea de alto nivel Abatir al Jugador. Esta tarea se descompone en Armarse, Apuntar y Disparar. Apuntar y Disparar son tareas atómicas que el agente puede llevar a cabo, pero Armarse debe descomponerse en acciones atómicas realizables por el agente: Coger Arma y Cargar Arma.

Un ejemplo de uso de planificadores jerárquicos está representado por el juego S.T.A.L.K.E.R.: Shadow of Chernobyl (GSC Game World / THQ, 2007) (vea figura 28.86), que usa jerarquías de planificadores GOAP de una manera muy particular.

Uno de los formalismos que aplican la planificación jerárquica son las redes jerárquicas de tareas (Hierarchical Task Networks o HTNs). Esta se define como *una colección de tareas que se desean que sean realizadas, junto con una serie de restricciones sobre el orden en que pueden llevarse a cabo, la forma en que se instancian sus variables y las condiciones que deben ser ciertas antes o después de que cada tarea se lleve a cabo*. Su uso en el desarrollo de videojuegos es cada vez es más extendido.

28.5.6. Reflexiones sobre el futuro

El algoritmo de búsqueda A* y la Máquina Finita de Estados (FSM) son probablemente las dos técnicas más empleadas de la IA en el desarrollo de videojuegos, para establecer caminos o rutas y para modelar el comportamiento de un personaje, respectivamente.

No obstante, la IA cada vez tiene más importancia en el desarrollo de videojuegos (vea la figura 28.88). Puesto que cada vez se podrá disponer de computación adicional, esto da pie a dedicar un procesador, o parte de él, a la IA y aplicar técnicas más avanzadas (Lógica Difusa, Computación Evolutiva, Redes Neuronales, Modelos Probabilísticos,...).

Figura 28.88: Cuando Resistance 3 fue presentado por Cameron Christian, de Insomniac Games, lo que señaló como la mayor mejora de la secuela era su IA mejorada.

El reto en el desarrollo de los videojuegos, es hacer que estos sean más naturales, más interactivos y más sociables. El uso de la IA en los videojuegos está en una fase de progreso significativo.

Por otra parte hay que indicar que los videojuegos son un gran reto para la IA y representan un campo de investigación en sí mismos, tratando de proponer métodos y técnicas que aporten un tipo de inteligencia que aunque no sea perfecta, si permita mejorar la experiencia de juego.

Hay que tener en cuenta que las situaciones manejadas por la IA son altamente dependientes del tipo juego y de sus requisitos de diseño, por lo que es posible que una solución utilizada en un videojuego no pueda ser aplicable en otro. Esto conduce al desarrollo de técnicas específicas. Por lo tanto, éste es un campo que ofrece un gran potencial en términos de investigación y desarrollo de nuevos algoritmos.

El objetivo de los videojuegos es desarrollar mundos virtuales, con el máximo parecido posible al mundo real, tanto a nivel visual como funcional y de interacción. En este sentido hay que destacar que en el mundo real es habitual el concepto de organización, es decir dos o más personas se juntan para cooperar entre sí y alcanzar objetivos comunes, que no pueden lograrse, o bien cuesta más lograrlos, mediante iniciativa individual.

En el campo del desarrollo de videojuegos este concepto deberá ser tratado, permitiendo a los personajes organizarse y cooperar para lograr un objetivo común, por ejemplo abatir al jugador o conseguir ganar una batalla o un partido. Por una parte, la organización implicará la existencia de una jerarquía y una distribución y especialización en tareas, y por otra la cooperación, implica la existencia de un mecanismo de comunicación común. Además, en el mundo real también es necesario facilitar la movilidad de profesionales, las organizaciones mejoran sus modos de trabajo gracias a la aportación de nuevos miembros que poseen mejor cualificación.

En el campo de los videojuegos, la movilidad implicaría permitir que personajes virtuales con un desempeño demostrado pudieran desplazarse de unas instacias de juego a otras, a través la red. Y esto último permite introducir otro aspecto de suma importancia, la capacidad de adaptabilidad o aprendizaje, un personaje mejorará su desempeño si tiene capacidad para aprender. Todo esto conduce a los sistemas multiagentes que tendrán mucho que aportar en los próximos años en el campo del desarrollo de videojuegos.

Los sistemas multiagentes proporcionan el marco donde un conjunto de agentes con tareas y habilidades claramente definidas colaboren y cooperen para la consecusión de sus objetivos. Existen multitud de metodologías y enfoques para el diseño de sistemas multiagentes así como plataformas para su desarrollo. Estas proporcionan soporte a los diseñadores y desarrolladores para construir sistemas fiables y robustos, pero seguro que será necesaria su adaptación para el mundo de los videojuegos.

28.6. Caso de estudio. Juego deportivo

El objetivo de la presente sección consiste en plantear el diseño básico para la Inteligencia Artificial de las distintas entidades que participan en un **juego deportivo de equipos**. Aunque la propuesta inicial que se discute es general y se podría reutilizar para una gran parte de los juegos deportivos, en esta sección se ha optado por estudiar el caso concreto del fútbol [16].

En el año 2050...

...se espera que un equipo de fútbol compuesto de robots físicos autónomos sea capaz de derrotar a un equipo de fútbol real.

El diseño y la implementación de la IA en un juego deportivo en equipo, particularmente en el fútbol, no es una tarea trivial debido a la complejidad que supone crear agentes autónomos capaces de jugar de una forma similar a la de un contrincante humano.

En este contexto, resulta interesante destacar la ***Robocup***[2], una competición que alberga torneos de fútbol entre equipos físicos de robots, por una parte, y entre equipos simulados mediante software, por otra, desde 1996.

Este último tipo de modalidad gira principalmente en torno al desarrollo de componentes de IA con el objetivo de dotar a los jugadores virtuales del mayor realismo posible y, eventualmente, derrotar al equipo virtual contrario.

La liga de simulación de fútbol (*Soccer Simulation League*) es una de las competeciones más antiguas de la *Robocup* y está centrada en la IA y las estrategias en equipo. Los jugadores virtuales o agentes se mueven de manera independiente en un terreno de juego virtual simulado en un computador. Dicha liga está compuesta a su vez de dos subligas, una simulada en el espacio 2D y otra en el espacio 3D. El software necesario para llevar a cabo simulaciones e incluso recrear la IA de los equipos virtuales ganadores de las distintas ediciones de estas ligas se puede descargar libremente de Internet[3].

Estos módulos de IA se han ido modificando hasta convertirse en sofisticados componentes software, los cuales rigen el comportamiento de los jugadores virtuales no sólo a nivel individual, sino también a nivel de equipo. Estos componentes se basan en algunas de las técnicas estudiadas en el presente capítulo, como por ejemplo la lógica difusa, y hacen uso de estructuras de datos no lineales, como los gráfos estudiados en el módulo 3, *Técnicas Avanzadas de Desarrollo*, para llevar a cabo una coordinación multi-agente.

Para facilitar el diseño y desarrollo de comportamientos inteligentes en el caso particular del fútbol, en este capítulo se discuten las bases de un ***framework*** que permite la integración y validación de comportamientos en un entorno 2D. De este modo, el lector podrá dedicar principalmente su esfuerzo a la creación y *testing* de dichos comportamientos con el objetivo de adquirir las habilidades necesarias para desarrollar la IA en un juego deportivo.

Evidentemente, este *framework* representa una versión muy simplificada de las reglas reales del fútbol con el objetivo de no incrementar la complejidad global del mismo y facilitar el desarrollo de comportamientos *inteligentes*.

Antes de pasar a discutir la arquitectura propuesta en la sección 28.6.2, relativa al *framework* que permitirá la implementación de comportamientos para los jugadores virtuales de un equipo de fútbol, en la siguiente sección se realiza una breve introducción a ***Pygame***, un conjunto de módulos de Python creados con el principal objetivo de programar juegos 2D. En este contexto, *Pygame* se ha utilizado para llevar a cabo la representación gráfica de dicho *framework*.

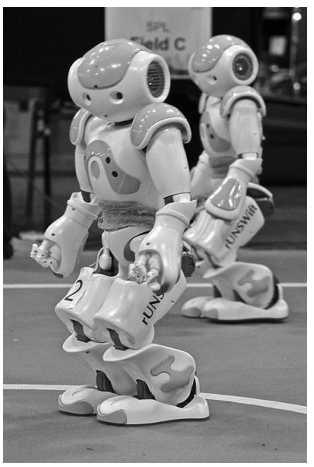


Figura 28.89: Ejemplo de robot compitiendo en la liga de Robocup de 2010 en Singapur (fuente Wikipedia).

28.6.1. Introducción a *Pygame*

Pygame[4] es un **conjunto de módulos** para el lenguaje de programación Python concebidos para el desarrollo de juegos. Básicamente, *Pygame* añade funcionalidad sobre la biblioteca multimedia SDL (estudiada en el módulo 1, *Arquitectura del Motor*, para el caso particular de la integración de sonido), abstrayendo la representación gráfica, la integración de sonido y la interacción con dispositivos físicos, entre otros aspectos, al desarrollador de videojuegos.

[2] `http://www.robocup.org/`
[3] `http://sourceforge.net/projects/sserver/`
[4] `http://pygame.org`

Figura 28.90: *Pygame* mantiene la filosofía de Python, simplicidad y flexibilidad a través de un conjunto de módulos de alto nivel para el desarrollo de juegos 2D.

Pygame es altamente portable y, por lo tanto, se puede ejecutar sobre una gran cantidad de plataformas y sistemas operativos. Además, mantiene una licencia LGPL[5], permitiendo la creación de *open source*, *free software*, *shareware* e incluso juegos comerciales.

A continuación se enumeran las **principales características** de *Pygame*:

- Gran nivel funcional, debido a la continuidad del proyecto.
- Independencia respecto a la biblioteca gráfica de bajo nivel, no siendo necesario utilizar exclusivamente OpenGL.
- Soporte multi-núcleo.
- Uso de código C optimizado y de ensamblador para la implementación de la funcionalidad del núcleo de *Pygame*.
- Alta portabilidad.
- Facilidad de aprendizaje.
- Amplia comunidad de desarrollo.
- Autonomía para controlar el bucle principal de juego.
- No es necesaria una interfaz gráfica para poder utilizar *Pygame*.
- Sencillez en el código, debido a la simplicidad de *Pygame*.

Al igual que ocurre en SDL, uno de los componentes fundamentales en *Pygame* está representado por el concepto de **superficie** (*surface*)[6], debido a que es el objeto contemplado por *Pygame* para la representación de cualquier tipo de imagen. En la tabla 28.7 se resume la lista de módulos que conforman *Pygame* junto a una breve descripción.

La instalación de *Pygame* en Debian GNU/Linux es trivial mediante los siguientes comandos:

```
$ sudo apt-get update
$ sudo apt-get install python-pygame
```

El siguiente listado de código muestra la posible implementación de una clase que encapsula la información básica de un campo de fútbol a efectos de renderizado (ver figura 28.91). Como se puede apreciar en el método constructor (*__init()__*) de la clase *SoccerField*, se crea una variable de clase, denominada *surface*, que representa la superficie de renderizado sobre la que se dibujarán las distintas primitivas que conforman el terreno de juego. Dicha superficie tiene una resolución determinada.

Listado 28.33: Clase SoccerField

```
1 # Incluir módulos de pygame...
2 TOP_LEFT = (50, 50)
3 WIDTH_HEIGHT = (1050, 720)
4
5 class SoccerField:
6
7     def __init__ (self, width, height):
8
```

[5] http://www.pygame.org/LGPL

[6] http://www.pygame.org/docs/ref/surface.html

```
 9         self.width, self.height = width, height
10
11         pygame.display.set_caption('CEDV - 2011/2012')
12         self.surface = pygame.display.set_mode(
13             (self.width, self.height), 0, 32)
14         # Terreno real de juego.
15         self.playing_area = Rect(TOP_LEFT, WIDTH_HEIGHT)
16
17     def render (self):
18
19         # Renderiza el terreno de juego.
20         self.surface.fill(THECOLORS['green']);
21         pygame.draw.rect(self.surface, THECOLORS['white'],
22                          self.playing_area, 7)
23         pygame.draw.circle(self.surface, THECOLORS['white'],
24                            self.playing_area.center, 10, 5)
25         pygame.draw.circle(self.surface, THECOLORS['white'],
26                            self.playing_area.center, 75, 5)
27         pygame.draw.line(self.surface, THECOLORS['white'],
28                          self.playing_area.midtop,
29                          self.playing_area.midbottom, 5)
```

Por otra parte, el método *render()* contiene las primitivas de *Pygame* necesarias para dibujar los aspectos básicos del terreno de juego. Note cómo se hace uso de las primitivas geométricas *rect*, *circle* y *line*, respectivamente, para dibujar las líneas de fuera y el centro del campo. Además de especificar el tipo de primitiva, es necesario indicar el color y, en su caso, el grosor de la línea.

Típicamente, el método *render()* se ejecutará tan rápidamente como lo permitan las prestaciones hardware de la máquina en la que se ejecute el ejemplo, a no ser que se establezca un control explícito del número de imágenes renderizadas por segundo.

El siguiente listado de código muestra cómo inicializar *Pygame* (línea 14) e instanciar el terreno de juego (línea 17). El resto de código, es decir, el bucle infinito permite procesar los eventos recogidos por *Pygame* en cada instante de renderizado. Note cómo en cada iteración se controla si el usuario ha cerrado la ventana gráfica, capturado mediante el evento *QUIT*, y se lleva a cabo el renderizado del terreno de juego (línea 27) junto con la actualización de *Pygame* (línea 29).

La gestión de la tasa de *frames* generados por segundo es simple mediante la clase *Clock*. De hecho, sólo es necesario especificar la tasa de *frames* deseada mediante la función *clock.tick(FPS)* para obtener una tasa constante.

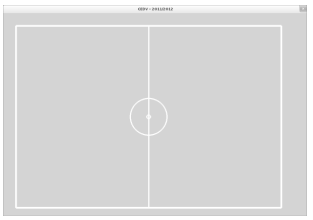


Figura 28.91: Resultado del renderizado de un terreno de juego simplificado mediante *Pygame*.

Event-oriented

La combinación de un esquema basado en eventos y una arquitectura con varias capas lógicas facilita la delegación y el tratamiento de eventos.

Listado 28.34: Ejecutando el ejemplo

```
1  # Incluir módulos de pygame...
2  WIDTH, HEIGHT = 1200, 800
3  FPS = 30
4
5  if __name__ == "__main__":
6      pygame.init()
7
8      clock = pygame.time.Clock()
9      soccerField = SoccerField(WIDTH, HEIGHT)
10
11     while True:
12         tick_time = clock.tick(FPS)
13
14         for event in pygame.event.get():
15             if event.type == QUIT:
16                 pygame.quit()
17                 sys.exit()
18
19         soccerField.render()
20         pygame.display.update()
```

Módulo	**Breve descripción**
camera	Soporte para el uso de cámaras
cdrom	Manejo de unidades de CD/DVD
color	Representación de colores
cursors	Soporte y gestión de cursores
display	Control sobre dispositivos y ventanas
draw	Renderizado de formas sobre una superficie
event	Interacción con eventos
examples	Módulo con ejemplos básicos
font	Carga y renderizado de fuentes
gfxdraw	Renderizado de formas sobre una superficie
image	Carga y almacenamiento de imágenes
joystick	Interacción con *joysticks*
key	Interacción con el teclado
locals	Constantes usadas por *Pygame*
mask	Soporte de máscaras sobre imágenes
midi	Interacción con dispositivos *midi*
mixer	Carga y reproducción de sonido
mouse	Interacción con el ratón
movie	Carga de vídeo *mpeg*
music	Control de *streaming* de audio
overlay	Soporte para *overlays* gráficos
pixelarray	Acceso directo a píxeles en superficies
pygame	Paquete de alto nivel de *Pygame*
rect	Objeto para gestionar rectángulos
scrap	Soporte para *clipboards*
sndarray	Acceso a *samples* de sonido
sprite	Módulo con objetos básicos
surface	Objeto para representar imágenes
surfarray	Acceso a píxeles de superficies mediante arrays
tests	Paquete para pruebas unitarias
time	Monitorización de tiempo
transform	Transformación de superficies

Cuadro 28.7: Resumen de los módulos que conforman *Pygame*.

28.6.2. Arquitectura propuesta

Simplificación

Recuerde que la simplicidad de una solución facilita la mantenibilidad del código y el desarrollo de nuevos componentes basados en los previamente desarrollados.

Como se ha comentado anteriormente, en este caso de estudio se ha llevado a cabo una simplificación de las reglas del fútbol con el objetivo de facilitar la implementación y permitir focalizar el esfuerzo en la parte relativa a la IA. Por ejemplo, se ha obviado la situación de fuera de juego pero sí que se contempla el saque de banda cuando el balón sale del terreno de juego. La figura 28.92 muestra de manera gráfica el estado inicial del juego, con un equipo a la izquierda del centro del campo y otro a la derecha.

A continuación, en esta subsección se discutirá el diseño de la versión inicial del *framework* propuesto [16]. Más adelante se prestará especial atención a cómo incluir nuevos tipos de comportamiento asociados a los propios jugadores.

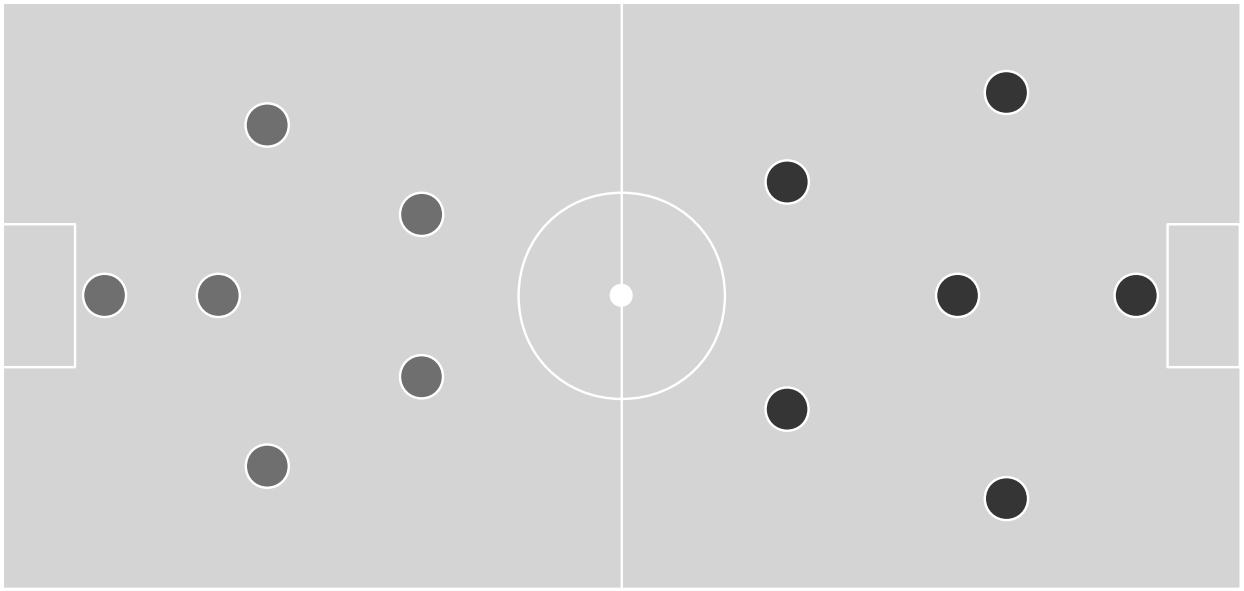


Figura 28.92: Terreno de juego con la posición inicial de los jugadores.

La figura 28.93 muestra, desde un punto de vista de alto nivel, el **diagrama de clases** de la arquitectura propuesta. Como se puede apreciar, la clase *SoccerField* aparece en la parte superior de la figura y está compuesta de dos equipos (*SoccerTeam*), dos porterías (*SoccerGoal*), un balón (*SoccerBall*) y un determinado número de regiones (*SoccerRegion*).

Por otra parte, el equipo de fútbol está compuesto por un determinado número de jugadores, cuatro en concreto, modelados mediante la clase *SoccerPlayer*, que a su vez mantiene una relación de herencia con la clase *MovingEntity*. A continuación se realizará un estudio más detallado de cada una de estas clases.

El terreno de juego

Una de las clases principales de la arquitectura propuesta es la clase **SoccerField**, la cual mantiene como estado interno referencias a los elementos más importantes que forman parten de dicha arquitectura. Estos elementos son los siguientes:

- **surface**, de tipo *pygame.surface*, que representa la superficie de *Pygame* utilizada para renderizar cada uno de los elementos que forman parte del juego.
- **playing_area**, de tipo *pygame.rect*, que representa el terreno real de juego considerando las líneas de fuera de banda y de córner.
- **walls**, de tipo *lista*, que contiene cuatro instancias de la clase *Wall2d*, las cuales representan las dos bandas y las dos líneas de córner. Estas estructuras se utilizan para detectar cuando el balón sale del terreno de juego.
- **regions**, de tipo *lista*, que contiene estructuras del tipo *SoccerRegion* que facilitan la división del terreno de juego en distintas zonas para gestionar aspectos como la posición inicial de los jugadores o la confección de estrategias de equipo.

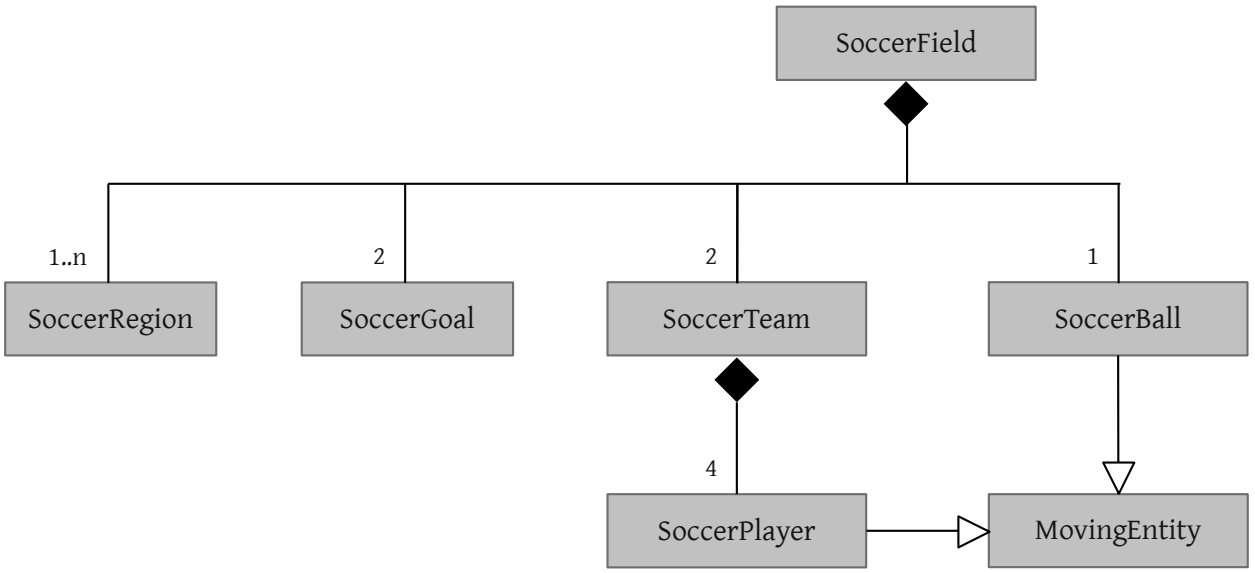


Figura 28.93: Diagrama de clases de la arquitectura del simulador de fútbol.

- **ball**, de tipo *SoccerBall*, que representa al balón usado para llevar a cabo las simulaciones.
- **teams**, de tipo *diccionario*, que permite indexar a dos estructuras de tipo *SoccerTeam* a partir del identificador del equipo (*red* o *blue*).
- **goals**, de tipo *diccionario*, que permite indexar a dos estructuras de tipo *SoccerGoal* a partir del identificador del equipo (*red* o *blue*).

El siguiente listado de código muestra la parte relativa a la inicialización de la clase *SoccerField*. Note cómo en el método constructor (*__init()__*) se lleva a cabo la instanciación de los elementos principales que conforman la simulación. Como se estudiará más adelante, algunas de estos elementos mantienen relaciones de agregación con otros elementos, como por ejemplo ocurre entre la clase *SoccerTeam* y la clase *SoccerPlayer* (un equipo está compuesto por varios jugadores).

Inicialización

Recuerde inicializar completamente el estado de una instancia en el proceso de creación. En el caso de Python, el constructor está representado por el método *__init()__*.

Además de coordinar la creación de las principales instancias del juego, la clase *SoccerField* también es responsable de la coordinación de la parte específica de renderizado o dibujado a través del método *render()*. Este método se expone en el siguiente listado de código.

Entre las líneas 9-17 se puede apreciar cómo se hace uso de ciertas primitivas básicas, ofrecidas por *Pygame*, para renderizar el terreno de juego, las líneas de fuera de banda y el centro del campo. Posteriormente, el renderizado del resto de estructuras de datos que forman parte de la clase se delega en sus respectivos métodos *render()*, planteando así un esquema escalable y bien estructurado. La actualización del renderizado se hace efectiva mediante la función *update()* de *Pygame* (línea 26).

Listado 28.35: Clase SoccerField. Método constructor

```
1 #!/usr/bin/python
2 # -*- coding: utf-8 -*-
```

```
3
4  class SoccerField:
5
6      def __init__ (self, width, height):
7
8          self.width, self.height = width, height
9
10         self.surface = pygame.display.set_mode(
11             (self.width, self.height), 0, 32)
12         # Terreno real de juego.
13         self.playing_area = Rect(TOP_LEFT, WIDTH_HEIGHT)
14
15         # Walls.
16         self.walls = []
17         # Cálculo de coordenadas clave del terreno de juego.
18         self.walls.append(Wall2d(top_left, top_right))
19         self.walls.append(Wall2d(top_right, bottom_right))
20         self.walls.append(Wall2d(top_left, bottom_left))
21         self.walls.append(Wall2d(bottom_left, bottom_right))
22
23         # Zonas importantes en el campo.
24         self.regions = {}
25         # Cálculo de las regiones...
26
27         # Bola.
28         self.ball = SoccerBall(Vec2d(self.playing_area.center),
29                                10, 2, Vec2d(0, 0), self)
30
31         # Equipos
32         self.teams = {}
33         self.teams['red'] = SoccerTeam('red', RED, 4, self)
34         self.teams['blue'] = SoccerTeam('blue', BLUE, 4, self)
35
36         # Porterías.
37         self.goals = {}
38         self.goals['red'] = SoccerGoal(
39             'red', RED, self.playing_area.midleft, self)
40         self.goals['blue'] = SoccerGoal(
41             'blue', BLUE, self.playing_area.midright, self)
```

En la figura 28.94 se muestra la división lógica del terreno de juego en una serie de **regiones** con el objetivo de facilitar la implementación del juego simulado, los comportamientos inteligentes de los jugadores virtuales y el diseño de las estrategias en equipo.

Internamente, cada región está representada por una instancia de la clase *SoccerRegion*, cuya estructura de datos más relevante es un rectángulo que define las dimensiones de la región y su posición.

Aunque cada jugador conoce en cada momento su posición en el espacio 2D, como se discutirá más adelante, resulta muy útil dividir el terreno de juego en regiones para implementar la estrategia interna de cada jugador virtual y la estrategia a nivel de equipo. Por ejemplo, al iniciar una simulación, y como se muestra en la figura 28.94, cada jugador se posiciona en el centro de una de las regiones que pertenecen a su campo. Dicha posición se puede reutilizar cuando algún equipo anota un gol y los jugadores han de volver a su posición inicial para efectuar el saque de centro.

Más info...

El uso de información o estructuras de datos adicionales permite generalmente optimizar una solución y plantear un diseño que sea más sencillo.

Listado 28.36: Clase SoccerField. Método render

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  class SoccerField:
5
6      def render (self):
7
```

```
        # Renderiza el terreno de juego.
        self.surface.fill(GREEN)
        pygame.draw.rect(self.surface, WHITE, self.playing_area, 3)
        pygame.draw.circle(self.surface, WHITE,
                           self.playing_area.center, 10, 2)
        pygame.draw.circle(self.surface, WHITE,
                           self.playing_area.center, 75, 2)
        pygame.draw.line(self.surface, WHITE,
                         self.playing_area.midtop,
                         self.playing_area.midbottom, 2)

        for g in self.goals.values(): # Porterías.
            g.render()
        for t in self.teams.values(): # Equipos.
            t.render()
        self.ball.render()            # Balón.

        # Actualización pygame.
        pygame.display.update()
```

La región *natural* de un jugador también puede variar en función de la estrategia elegida. Por ejemplo, un jugador que haga uso de un comportamiento defensivo puede tener como meta principal ocupar regiones cercanas a su portería, con el objetivo de evitar un posible gol por parte del equipo contrario.

El balón de juego

Una de las entidades más relevantes para recrear la simulación de un juego deportivo es la bola o balón, ya que representa la herramienta con la que los jugadores virtuales interactúan con el objetivo de ganar el juego. Esta entidad se ha modelado mediante la clase **SoccerBall**, la cual hereda a su vez de la clase *MovingEntity*, como se puede apreciar en el siguiente listado de código. Recuerde que el caso de estudio discutido en esta sección se enmarca dentro de un mundo virtual bidimensional.

Desde un punto de vista interno, el balón ha de almacenar la posición actual, la posición del instante de tiempo anterior y la información relevante del terreno de juego, es decir, la información relativa a las líneas de fuera de banda y de córner. El resto de información la hereda directamente de la clase *MovingEntity*, como por ejemplo la velocidad del balón, la velocidad máxima permitida o la dirección.

Una instancia de la clase *SoccerBall* se puede entender como una **entidad pasiva**, ya que serán los propios jugadores o el modelo básico de simulación física los que actualicen su velocidad y, en consecuencia, su posición. Por ejemplo, cuando un jugador golpee el balón, en realidad lo que hará será modificar su velocidad. Esta información se utiliza a la hora de renderizar el balón en cada instante de tiempo.

Listado 28.37: Clase SoccerBall. Método constructor

```
class SoccerBall (MovingEntity):

    def __init__ (self, pos, size, mass, velocity, soccer_field):

        MovingEntity.__init__(
            self, pos, size, velocity, velocity,
            Vec2d(soccer_field.playing_area.center), mass)

        self.oldPos = pos
        self.soccer_field = soccer_field
        self.walls = soccer_field.walls

class MovingEntity:

    def __init__ (self, pos, radius, velocity, max_speed,
```

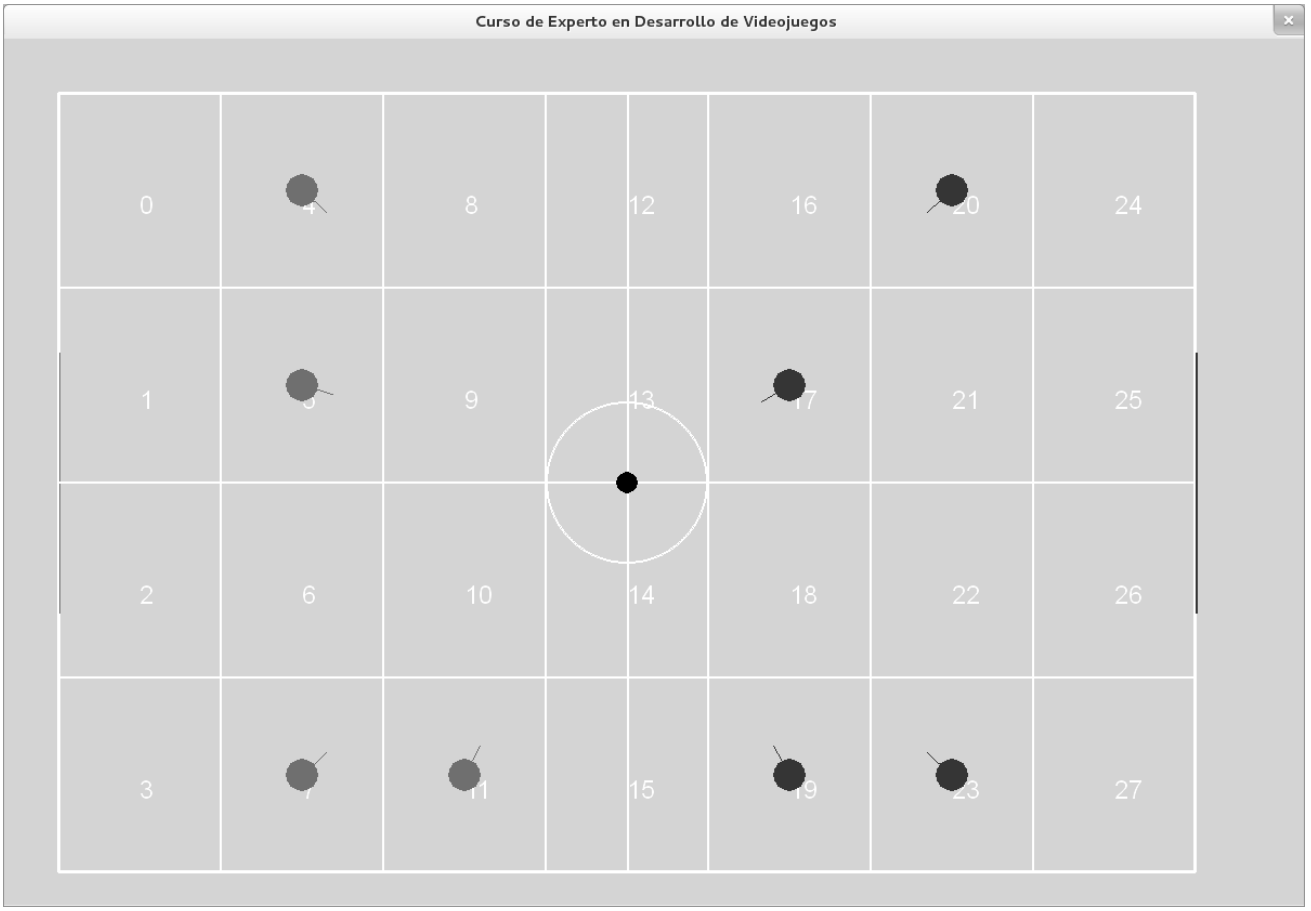


Figura 28.94: División del terreno de juego en regiones para facilitar la integración de la IA.

```
16                      heading, mass):
17
18          self.pos = pos
19          self.radius = radius
20          self.velocity = velocity
21          self.max_speed = max_speed
22          self.heading = heading
23          # Vector perpendicular al de dirección (heading).
24          self.side = self.heading.perpendicular
25          self.mass = mass
```

Dentro de la clase *SoccerBall* existen métodos que permiten, por ejemplo, detectar cuándo el balón sale fuera del terreno de juego. Para ello, es necesario comprobar, para cada uno de los laterales, la distancia existente entre el balón y la línea de fuera. Si dicha distancia es menor que un umbral, entonces el balón se posicionará en la posición más cercana de la propia línea para efectuar el saque de banda.

Listado 28.38: Clase SoccerBall. Método testCollisionWithWalls

```
1 def testCollisionWithWalls (self):
2
3     # Iterar sobre todos los laterales del campo
4     # para calcular si el balón interseca.
```

```
5       for w in self.walls:
6           if distToWall(w.a, w.b, self.pos) < TOUCHLINE:
7               self.reset(self.pos)
```

Por otra parte, el balón sufrirá el efecto del golpeo por parte de los jugadores virtuales mediante el método *kick()*. Básicamente, su funcionalidad consistirá en modificar la velocidad del balón en base a la fuerza del golpeo y la dirección del mismo.

Listado 28.39: Clase SoccerBall. Método kick

```
1 # Golpea el balón en la dirección dada.
2 def kick (self, direction, force):
3
4     # Normaliza la dirección.
5     direction = direction.normalized()
6     # Calculo de la aceleración.
7     acceleration = (direction * force) / self.mass
8     # Actualiza la velocidad.
9     self.velocity = acceleration
```

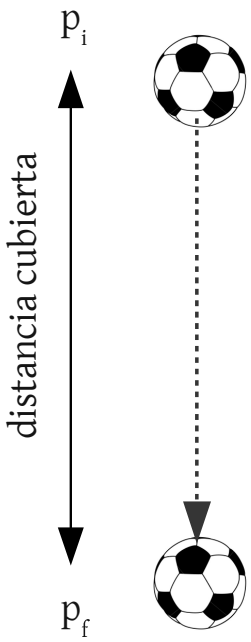


Figura 28.95: Esquema gráfico de la predicción del balón tras un instante de tiempo.

También resulta muy interesante incluir la funcionalidad necesaria para que los jugadores puedan prever la futura posición del balón tras un intervalo de tiempo. Para llevar a cabo el cálculo de esta futura posición se ha de obtener la distancia recorrida por el balón en un intervalo de tiempo Δt, utilizando la ecuación 28.7,

$$\Delta x = u\Delta t + \frac{1}{2}a\Delta t^2 \tag{28.7}$$

donde Δx representa la distancia recorrida, u es la velocidad del balón cuando se golpea y a es la deceleración debida a la fricción del balón con el terreno de juego.

Tras haber obtenido la distancia a recorrer por el balón, ya es posible usar esta información para acercar el jugador al balón. Sin embargo, es necesario hacer uso de la dirección del propio balón mediante su vector de velocidad. Si éste se normaliza y se multiplica por la distancia recorrida, entonces se obtiene un vector que determina tanto la distancia como la dirección. Este vector se puede añadir a la posición del balón para predecir su futura posición.

Este tipo de información es esencial para implementar estrategias de equipo y planificar el movimiento de los jugadores virtuales. Dicha funcionalidad, encapsulada en la clase *SoccerBall*, se muestra en el siguiente listado de código.

Listado 28.40: Clase SoccerBall. Método futurePosition

```
1  # Devuelve la posición del balón en el futuro.
2  def futurePosition (self, time):
3
4      # u = velocidad de inicio.
5      # Cálculo del vector ut.
6      ut = self.velocity * time
7
8      # Cálculo de 1/2*a*t*t, que es un escalar.
9      half_a_t_squared = 0.5 * FRICTION * time * time
10
11     # Conversión del escalar a vector,
12     # considerando la velocidad (dirección) de la bola.
13     scalarToVector = half_a_t_squared * self.velocity.normalized()
14
15     # La posición predicha es la actual
16     # más la suma de los dos términos anteriores.
17     return self.pos + ut + scalarToVector
```

El jugador virtual

La entidad base para representar a cualquier jugador de fútbol virtual está representada por la clase *SoccerPlayer*, la cual a su vez es una especialización de la clase *MovingEntity* (al igual que el balón de juego). Esta entidad base se podrá extender para especializarla en función de distintos tipos de jugadores, típicamente jugadores de campo y portero.

El siguiente listado de código muestra una posible implementación básica de esta clase. Note cómo en el método *update()* se actualiza la velocidad interna del jugador en función de los comportamientos que tenga activos, tal y como se discute en la siguiente sección.

Listado 28.41: Clase SoccerPlayer

```
class SoccerPlayer (MovingEntity):

    def __init__ (self, team, colour, number, pos, soccer_field):
        # Inicialización de MovingEntity.
        self.team = team
        self.colour = colour
        self.number = number
        self.initialPos = pos
        self.soccer_field = soccer_field
        aux = (soccer_field.playing_area.center - pos).normalized()
        self.direction = aux

        # Comportamientos asociados.
        self.steeringBehaviours = SteeringBehaviours(
            self, soccer_field.ball)

    # Posición inicial (e.g. kickoff).
    def reset (self):
        self.pos = self.initialPos

    # Actualización de la posición del jugador.
    def update (self):
        self.velocity = self.steeringBehaviours.calculate()
        self.pos += self.velocity
```

Precisamente, estos comportamientos son los que determinan las habilidades del jugador y de su implementación dependerá en gran parte el resultado de una simulación.

28.6.3. Integración de comportamientos básicos

La figura 28.98 muestra el diagrama de clases con las distintas entidades involucradas en la integración de IA dentro del simulador futbolístico. La parte más relevante de este diagrama está representado por la clase *SteeringBehaviours*, la cual aglutina los distintos comportamientos asociados, desde un punto de vista general, al jugador de fútbol. Note también cómo la clase base asociada al jugador de fútbol, *SoccerPlayer*, ha sido especializada en otras dos clases, *FieldPlayer* y *Goalkeeper*, con el objetivo de personalizar el comportamiento de los jugadores de campo y el portero, respectivamente.

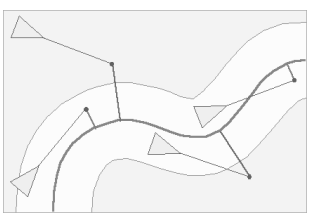


Figura 28.96: Seguimiento de un camino por una entidad basada en un modelo de comportamientos badados en direcciones.

El modelo de comportamientos usado en esta sección se basa en el modelo[7] presentado originalmente por C.W. Reynolds en el *Game Developers Conference* de 1999 [74].

Básicamente, este modelo se basa en la **combinación de múltiples comportamientos** para navegar en un espacio físico de una manera *improvisada*, es decir, guiando el movimiento de la entidad en cuestión en base a las condiciones del entorno en cada instante y en la posición de la propia entidad. Estos comportamientos se plantean de manera independiente a los medios de locomoción de la entidad o el carácter móvil.

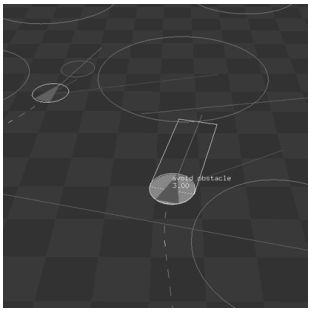


Figura 28.97: Captura de pantalla de un ejemplo en ejecución de la biblioteca *OpenSteer*.

El modelo de *Steering Behaviours* trata de facilitar el diseño de situaciones como la siguiente: *ir de A a B mientras se evitan obstáculos y se viaja junto a otras entidades*.

Existen distintas herramientas y bibliotecas que facilitan la integración de este tipo de comportamientos en problemas reales, como es el caso de la biblioteca *OpenSteer*[8], una implementación en C++ que facilita el uso, la combinación y la construcción de este tipo de comportamientos.

Este tipo de bibliotecas suelen integrar algún mecanismo de visualización para facilitar la depuración a la hora de implementar comportamientos. En el caso de *OpenSteer* se incluye una aplicación denominada *OpenSteerDemo* que hace uso de OpenGL para llevar a cabo dicha visualización, como se puede apreciar en la figura 28.97.

Se recomienda al lector la descarga y estudio de *OpenSteer* como ejemplo particular de biblioteca para el estudio e implementación de diversos comportamientos sobre entidades virtuales móviles.

Retomando el diagrama de clases de la figura 28.98, resulta importante profundizar en la implementación de la clase **SteeringBehaviours**, la cual representa la base para la integración de nuevos comportamientos. Básicamente, la idea planteada en esta clase se basa en los siguientes aspectos:

- Cada comportamiento tiene como objetivo calcular la fuerza que se ha de aplicar al objeto que lo implementa, en el espacio 2D, para materializar dicho comportamiento.
- Los comportamientos son independientes entre sí, siendo necesario incluir algún tipo de esquema para combinarlos. Por ejemplo, es posible incluir un esquema basado en prioridades o **pesos** para obtener la fuerza global a aplicar a un objeto.
- La implementación planteada es escalable, siendo necesario implementar un nuevo método en la clase *SteeringBehaviours* para incluir un nuevo comportamiento.

[7] `http://www.red3d.com/cwr/steer/`
[8] `http://opensteer.sourceforge.net/`

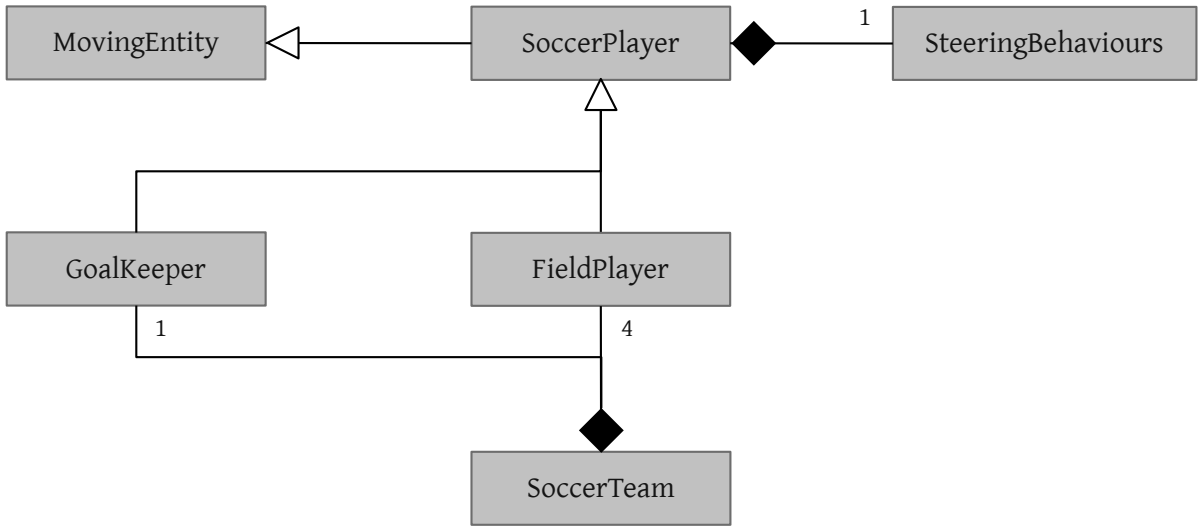


Figura 28.98: Diagrama de clases del modelo basado en comportamientos para la simulación de fútbol.

El siguiente listado de código muestra el método constructor de la clase *SteeringBehaviours*. Como se puede apreciar, dicha clase incluye como estado un diccionario, denominado *activated*, con los distintos tipos de comportamientos habilitados en la misma que, inicialmente, no están activados.

Listado 28.42: Clase SteeringBehaviours. Método constructor

```
class SteeringBehaviours:

    def __init__ (self, player, ball):

        self.player = player
        self.ball = ball

        self.target = ball

        # Comportamientos activados.
        self.activated = {}
        self.activated['seek'] = False
        self.activated['pursuit'] = False
        self.activated['arrive'] = False
        # Más comportamientos aquí...
```

Típicamente será necesario activar varios comportamientos de manera simultánea. Por ejemplo, en el caso del simulador de fútbol podría ser deseable activar el comportamiento de perseguir el balón pero, al mismo tiempo, recuperar una posición defensiva. En otras palabras, en un determinado instante de tiempo un jugador virtual podría tener activados los comportamientos de *pursuit* (perseguir) y *arrive* (llegar).

Desde el punto de vista de la implementación propuesta, la instancia del jugador virtual tendría que activar ambos comportamientos mediante el diccionario denominado *activated*. Posteriormente, sería necesario calcular la fuerza resultante asociada a ambos comportamientos en cada instante de tiempo para ir actualizando la fuerza a aplicar y, en consecuencia, la posición del objeto virtual.

En el siguiente listado de código se muestra la implementación de los métodos *calculate()*, *sumForces()* y *truncate*, utilizados para devolver la **fuerza global** resultante, calcularla en base a los comportamientos activados y truncarla si así fuera necesario, respectivamente.

Listado 28.43: Clase SteeringBehaviours. Cálculo de fuerza

```
# Devuelve la fuerza global resultante
# en base a los comportamientos activados.
def calculate (self):

    steeringForce = self.sumForces()
    return steeringForce

# Habría que incluir un esquema basado en
# prioridades o pesos para combinar de
# manera adecuada los comportamientos.
def sumForces (self):

    force = Vec2d(0, 0)

    if self.activated['seek']:
        force += self.seek(self.target)
    if self.activated['pursuit']:
        force += self.pursuit(self.target)
    if self.activated['arrive']:
        force += self.arrive(self.target)
    # Más comportamientos aquí...

    return force

# Trunca a una fuerza máxima.
def truncate (self, max_force):

    if self.steeringForce > max_force:
        self.steeringForce = max_force
```

Como se ha comentado anteriormente, la inclusión de un nuevo comportamiento consiste en implementar un nuevo método dentro de la clase *SteeringBehaviours* y registrar la entrada asociada al mismo en el diccionario *activated*, con el objetivo de que un jugador virtual pueda activarlo o desactivarlo en función del estado de juego actual. El siguiente listado muestra la implementación de los comportamientos *seek* (buscar) y *pursuit* (perseguir).

Ambos comportamientos son esenciales para llevar a cabo la construcción de otros comportamientos más complejos. Note cómo la implementación se basa en geometría 2D básica para, por ejemplo, calcular la velocidad vectorial resultante como consecuencia de orientar a una entidad en una dirección determinada.

Listado 28.44: Clase SteeringBehaviours. Seek y pursuit

```
# Dado un objetivo, este comportamiento devuelve la fuerza
# que orienta al jugador hacia el objetivo y lo mueve.
def seek (self, target):

    desiredVelocity = (target - self.player.pos).normalized()
    desiredVelocity *= self.player.max_speed

    return (desiredVelocity - self.player.velocity)

# Crea una fuerza que mueve al jugador hacia la bola.
def pursuit (self, target):

    toBall = self.ball.pos - self.player.pos
    self.direction = toBall.normalized()
```

```
15      lookAheadTime = 0.0
16
17      if self.ball.velocity.get_length() != 0.0:
18          sc_velocity = self.ball.velocity.get_length()
19          lookAheadTime = toBall.get_length() / sc_velocity
20
21      # ¿Dónde estará la bola en el futuro?
22      target = self.ball.futurePosition(lookAheadTime)
23
24      # Delega en el comportamiento arrive.
25      return self.arrive(target)
```

28.6.4. Diseño de la Inteligencia Artificial

El esquema básico planteado hasta ahora representa la base para la implementación del módulo de IA asociado a los distintos jugadores del equipo virtual. En otras palabras, el código fuente base discutido hasta ahora se puede entender como un *framework* sobre el que construir **comportamientos *inteligentes*** a nivel de jugador individual y a nivel de estrategia de equipo.

Por ejemplo, la figura 28.99 muestra un ejemplo clásico en simuladores futbolísticos donde un jugador que no mantiene el balón en un determinado instante de tiempo, es decir, un *jugador de apoyo* (*supporting player*) ha de emplazarse en una buena posición para recibir el balón. Evidentemente, es necesario garantizar que no todos los jugadores de un equipo que no mantienen el balón se muevan a dicha posición, sino que han de moverse a distintas posiciones para garantizar varias alternativas de pase.

En el ejemplo de la figura 28.99, el jugador 2 representa una mejor alternativa de pase frente al jugador 3. Para evaluar ambas alternativas es necesario establecer un mecanismo basado en **puntuaciones** para computar cuál de ellas es la mejor (o la *menos mala*). Para ello, se pueden tener en cuenta los aspectos como los siguientes:

- Facilidad para llevar a cabo el pase sin que el equipo contrario lo intercepte. Para ello, se puede considerar la posición de los jugadores del equipo rival y el tiempo que tardará el balón en llegar a la posición deseada.
- Probabilidad de marcar un gol desde la posición del jugador al que se le quiere pasar.
- Histórico de pases con el objetivo de evitar enviar el balón a una posición que ya fue evaluada recientemente.

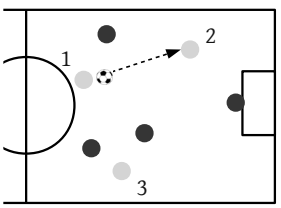


Figura 28.99: Esquema de apoyo de jugadores. El jugador 2 representa una *buena* alternativa mientras que el jugador 3 representa una *mala*.

La figura 28.100 muestra de manera gráfica el cálculo de la puntuación de los diversos puntos de apoyo al jugador que tenga el balón con el objetivo de llevar a cabo el siguiente pase. El radio de cada uno de los puntos grises representa la puntuación de dichos puntos y se pueden utilizar para que el jugador de apoyo se mueva hacia alguno de ellos.

El estado de juego determina en gran parte el estado interno de los jugadores, ya que no se debería actuar del mismo modo en función de si el equipo ataca o defiende. Desde el **punto de vista defensivo**, una posible alternativa consiste en retrasar la posición de los jugadores de un equipo con el objetivo de dificultar los pases a los jugadores del equipo contrario en posiciones cercanas a la portería.

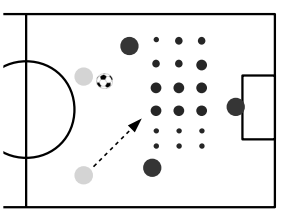


Figura 28.100: Esquema gráfico relativo al cálculo de los mejores puntos de apoyo para recibir un pase.

Para ello, se puede establecer un sistema defensivo basado en posicionar a los jugadores en ciertas zonas o regiones clave del terreno de juego, utilizando para ello la información espacial de la figura 28.94. En este contexto, se pueden definir una serie de posiciones iniciales o *home positions* a las que los jugadores se replegarán cuando haya una pérdida de balón.

Esta estrategia también puede facilitar el contraataque cuando se recupera el balón. Evidentemente, esta estrategia defensiva no es la única y es posible establecer un esquema defensivo que sea más agresivo. Por ejemplo, se podría plantear un esquema basado en la presión por parte de varios jugadores al jugador del equipo rival que mantiene el balón. Sin embargo, este esquema es más arriesgado ya que si el jugador rival es capar de efectuar un pase, sobrepasando a varios jugadores del equipo que defiende, la probabilidad de anotar un tanto se ve incrementada sustancialmente.

La figura 28.101 muestra un ejemplo concreto de posicionamiento de los jugadores del equipo que defiende en base a regiones clave del terreno de juego. Este mismo esquema se puede utilizar cuando el jugador contrario realice un saque de centro.

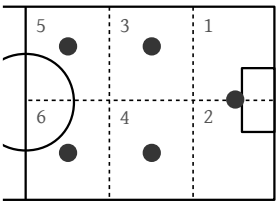


Figura 28.101: Esquema defensivo basado en la colocación de los jugadores del equipo que defiende en determinadas regiones clave.

28.6.5. Consideraciones finales

En esta sección se ha planteado una implementación inicial de un simulador de fútbol con el objetivo de proporcionar un enfoque general relativo a este tipo de problemática. A partir de aquí, es necesario prestar especial atención a los siguientes aspectos para incrementar la funcionalidad del prototipo discutido:

- La utilización de la información básica de los jugadores, especialmente la posición, la velocidad y la dirección actuales, junto con el estado del balón, permite integrar una gran cantidad de funcionalidad asociada a los jugadores, desde la interceptación de pases hasta esquemas de combinación avanzados (e.g. *uno-dos* o *paredes*).
- El comportamiento interno a nivel de jugador individual y de equipo debería modelarse utilizando algún tipo de **máquina de estados**, como las discutidas en la sección 28.5.4. Así, sería posible conocer si, por ejemplo, un jugador ataca o defiende. Típicamente, la recuperación o pérdida del balón disparará una transición que condicionará el cambio del estado actual.
- El modelo basado en *Steering Behaviours* representa la base para añadir nuevos comportamientos. En los capítulos 3 y 4 de [16] se discuten estos aspectos más en profundidad prestando especial atención al simulador futbolístico.
- La funcionalidad de los jugadores virtuales se puede extender añadiendo complejidad e integrando técnicas más avanzadas, como por ejemplo el regate o *dribbling*.
- Es posible integrar técnicas como la Lógica Difusa, estudiada en la sección 28.2, para manejar la incertidumbre y, por lo tanto, establecer un marco más realista para el tratamiento de la problemática de esta sección.

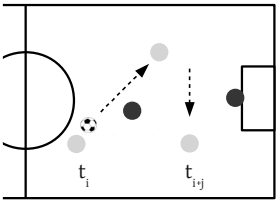


Figura 28.102: Esquema de combinación avanzada entre jugadores para sorter al jugador rival mediante *paredes*.

28.7. Sistemas expertos basados en reglas

28.7.1. Introducción

Los sistemas expertos son una rama de la Inteligencia Artificial simbólica que simula comportamientos humanos razonando con comportamientos implementados por el diseñador del software. Los sistemas expertos se usan desde hace más de 30 años, siendo unos de los primeros casos de éxito de aplicación práctica de técnicas de Inteligencia artificial. Existen diversos tipos: basados en casos, árboles de decisión, basados en reglas, etc. Este último tipo será el que se aplicará al desarrollo de inteligencia artificial para un juego a lo largo del presente apartado.

Los sistemas expertos basados en reglas (*rule-based expert systems* en inglés) están compuestos por los siguientes tres elementos que se observan en la figura 28.103:

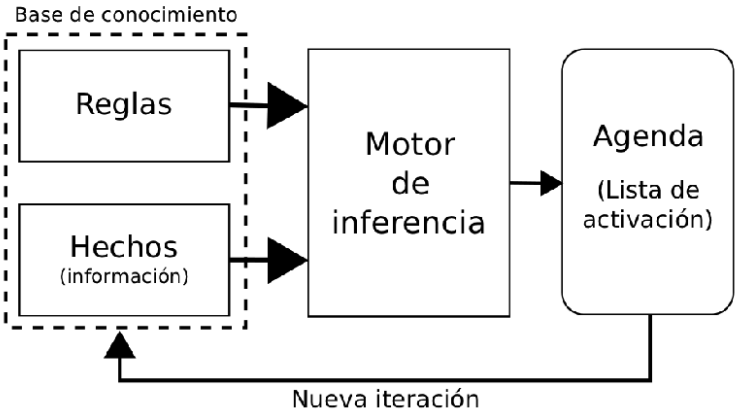


Figura 28.103: Arquitectura básica de un sistema experto basado en reglas.

Base de conocimiento incluye los *hechos*, información almacenada en el sistema, y las *reglas de inferencia*, que permiten generar nuevo conocimiento a partir de los hechos.

Motor de inferencia es el algoritmo encargado de analizar la satisfacibilidad de las reglas con los hechos presentes en el sistema.

Agenda es el conjunto de reglas del sistema activables en un momento dado. Debe incorporar una *estrategia de resolución de conflictos*, que es un algoritmo para decidir de qué regla activar en caso de que haya varias posibles. Algunos algoritmos típicos son la cola (ejecuta la primera regla activable que se definió en el sistema), aleatorio, activación de la regla menos usada, etc

El funcionamiento del sistema es sencillo: el motor de inferencia comienza evaluando las reglas activables inicialmente en el sistema (aquellas cuyas condiciones de activación se cumplen con los hecho iniciales), pasando a introducirlas en la agenda. De entre las reglas en la agenda, selecciona una de acuerdo a la estrategia de resolución de conflictos definida, y la ejecuta. Dicha ejecución (también llamado *disparo*, del inglés *trigger*) suele provocar cambios en los hechos del sistema. Por lo tanto, es necesario recalcular la agenda, pues habrá reglas que dejen de ser activables (pues alguna de sus condiciones ya no se cumpla) y otras que pasen a serlo (porque ya se cumplan todas sus condiciones). Tras este paso de nuevo se usa la estrategia de resolución de conflictos para elegir la regla a disparar, realizando una nueva iteración. El sistema continuará iterando de esta manera hasta que no tiene reglas que activar en la agenda.

Este proceso podría parecer ineficiente computacionalmente, sobre todo en términos de tiempo necesario para recorrer la lista de reglas evaluando condiciones. Sin embargo, como el conjunto de reglas (y sus condiciones) no cambian a lo largo de la vida del sistema, los entornos de ejecución suelen introducir diversas optimizaciones, siendo la más destacada la creación de una red de reglas que comparten nodos (condiciones).

Dado que el número de hechos que suele cambiar tras cada activación de una regla suele ser bajo, una adecuada organización de condicionantes permite, a nivel práctico, dar unos tiempos de ejecución adecuados. Si se desea ampliar información sobre este asunto, el principal libro de referencia es [39].

28.7.2. Caso de estudio: Stratego

Los conceptos presentados anteriormente se aplicarán a una versión simplificada del juego de tablero Stratego [103]. En el juego participan dos jugadores que tienen a su cargo dos ejércitos idénticos de 16 fichas, cada una con un valor. El objetivo es capturar la ficha más débil del rival, la única con valor 1. Las partidas se juegan en un cuadrante de 8x8 casillas, comenzando un ejército en las dos primeras filas del tablero y el otro en las dos últimas. Inicialmente los dos jugadores conocen las posiciones ocupadas por las fichas de los dos ejércitos, pero sólo conocen los valores concretos de sus fichas, que las coloca en el orden que considere más adecuado. Por lo tanto existe un conocimiento parcial del entorno como se observa en la figura 28.104, que muestra un estado inicial de una partida.

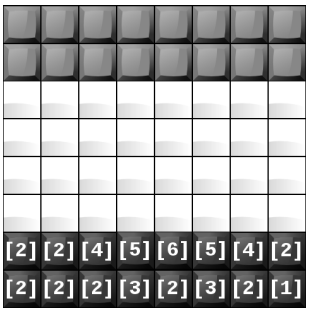


Figura 28.104: Posible estado inicial de una partida.

Los jugadores mueven una ficha propia a una casilla adyacente por turnos. Cada casilla sólo la puede ocupar una ficha, excepto las casillas *obstáculo*, que no las puede ocupar ficha alguna. En caso de que dos fichas de distinto ejército coincidan en la misma casilla la de menor valor queda capturada (eliminándose del tablero) y la de mayor valor se queda en la casilla siendo descubierta su puntuación al jugador contrario. En caso de que colisionen dos fichas de igual valor, se eliminan ambas. En la figura 28.105 se muestra un estado intermedio de la partida con fichas en posiciones distintas a las iniciales, algunas con su valor descubierto y otras capturadas no aparecen.

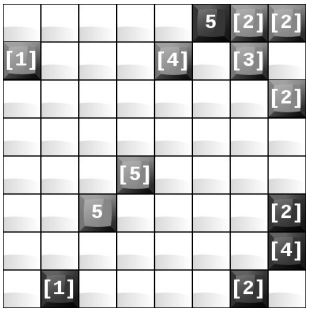


Figura 28.105: Posible estado intermedio de una partida.

Para implementar un sistema experto que pueda jugar a este juego se usará CLIPS[9], un entorno para el desarrollo de sistemas expertos basados en reglas que inicialmente fue desarrollado por la NASA y actualmente mantenido por uno de sus creadores, Gary Riley. CLIPS está disponible bajo licencia libre en su web [75], y se puede usar interactivamente desde consola o integrándolo con diversos lenguajes de programación (C/C++, Java, Python, etc).

En el listado 28.45 se puede observar diversos hechos de ejemplo. El primero (línea 3) nos indica que se está jugando el turno número 30 de la partida (duración por defecto de la partida es 200 turnos). El segundo hecho tiene dos campos (pos-x y pos-y), e informa que en la casilla $(4, 5)$ del tablero hay un obstáculo. El tercer hecho nos indica que en la esquina inferior izquierda del tablero hay a una ficha con identificador *111* y su valor es 3 puntos. Este valor lo desconoce el otro jugador porque el campo *descubierta* está a 0. El siguiente hecho (línea 10) nos informa que justo arriba de la ficha anterior se tiene otra con el identificador *929* y también de 3 puntos de valor, pero en esta ocasión su valor sí es conocido por el rival. Por último, en las dos lineas de abajo del todo se nos indica que dos fichas del jugador contrario (*equipo B*) rodeando la ficha *929*, una en la casilla $(1, 3)$ cuyo valor no se conoce y otra en la casilla $(2, 2)$ que vale 3 puntos.

Listado 28.45: Ejemplos de hechos del sistema

```
1 ; Esta linea es un comentario
2
3 (tiempo 30)
4
5 (obstaculo (pos-x 4) (pos-y 5))
```

[9]No confundir con Common LISP (CLISP), otro lenguaje cuya sintaxis también se basa en paréntesis anidados.

```
6
7  (ficha (equipo "A") (num 111) (pos-x 1) (pos-y 1)
8    (puntos 3) (descubierta 0))
9
10 (ficha (equipo "A") (num 929) (pos-x 1) (pos-y 2)
11   (puntos 3) (descubierta 1))
12
13 (ficha (equipo "B") (pos-x 1) (pos-y 3)
14        (descubierta 0)
15
16 (ficha (equipo "B") (pos-x 2) (pos-y 2) (puntos 3)
17        (descubierta 1))
```

En el listado 28.46 se puede observar una regla de ejemplo. La regla comienza con la palabra reservada *defrule* seguida de la cadena *EQUIPO-A::*, que indica que la regla pertenece al módulo de inteligencia artificial del ejército. A continuación aparece el nombre de la regla, que suele describir su intención (atacar en este caso). En la segunda línea se indica la prioridad de la regla con la palabra reservada *salience*. CLIPS permite reglas de prioridades desde 1 a 100, de modo que su estrategia de resolución de conflictos sólo ejecutará una regla de una prioridad determinada si no existen reglas de prioridad superior en la agenda. En el caso de este juego se usarán prioridades de 1 a 80 para las reglas de inteligencia artificial, dejando prioridades superiores para el control del *game loop*.

Las siguientes líneas hasta el símbolo $=>$ son las condiciones que deben cumplir los hechos del sistema para poder activar la regla. Cada campo de las condiciones puede tener bien una constante (en este caso la letra "A" en la línea 3, pero también se podría fijar la posición X o Y por ejemplo) o una variable. Las variables son cadenas de alfanuméricas sin espacios que comienzan por el símbolo *?*. En este caso concreto la regla busca una ficha propia cualquiera, sobre la que no se imponen restricciones a identificador, posiciones X e Y ni valor en puntos.

La línea después del símbolo $=>$ indica la acción a efectuar si la regla se activa. Hay que recordar que de todas las reglas activables en la agenda sólo se activa una en cada paso, pues su activación probablemente cambie hechos del sistema y como consecuencia la composición de la agenda). En nuestro caso se usa la palabra reservada *assert* para indicar que la ficha que se seleccionó anteriormente (porque se usa su mismo identificador, $?n1$) se mueva hacia arriba[10] en el turno actual (*tiempo ?t*).

Listado 28.46: Versión inicial de la regla atacar

```
1 (defrule EQUIPO-A::atacar
2    (declare (salience 30))
3    (ficha (equipo "A") (num ?n1) (pos-x ?x1) (pos-y ?y1)
4           (puntos ?p1))
5    (tiempo ?t)
6  =>
7    (assert (mueve (num ?n1) (mov 3) (tiempo ?t)))
8 )
```

A primera vista, esta regla puede parecer adecuada, al fin y al cabo sólo mueve una ficha hacia delante, que es donde están los contrarios al comenzar la partida. Y como su prioridad es bastante baja, sólo se ejecutará si no hay otra regla de mayor prioridad (que se supone hará una acción mejor). Sin embargo es muy mejorable. Por ejemplo,

[10]Los movimientos están definidos como 1 avanza en X, 2 retrocede en X, 3 avanza en Y y 4 retrocede en Y.

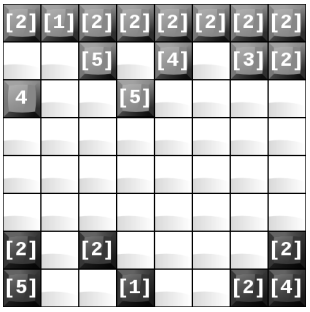


Figura 28.106: Estado de partida para ataque.

pudiera ser que la ficha se dirigiera hacia una ficha contraria de mayor puntuación, como podría ser la ficha de 2 puntos del equipo púrpura que está en la primera fila del tablero en la figura 28.106. Si se aplica la regla anterior a dicha ficha se estarían perdiendo varios turnos para suicidarla.

Así que se podría cambiar la regla por la que aparece en el listado 28.47. En este caso se han añadido dos condiciones nuevas. La primera (línea 5) busca la existencia de una ficha contraria (*equipo "B"*) en la misma columna (pues es usa la misma variable *?x1* para los dos campos *pos-x* en su línea 5). Y la segunda condición (línea 7) comprueba que la ficha contraria esté arriba de la propia y que tenga menos valor que esta. Nótese el uso de notación prefija, primero aparece el operador y después los operandos.

Puede observarse que la primera vez que aparece una variable se instancia al valor de un hecho del sistema, y a partir de ahí el resto de apariciones en esa y otras condiciones ya se considera ligada a una constante. En caso de que se cumplan todas las condiciones, la regla se considera activable. Si alguna de las condiciones no puede satisfacerse, se intenta satisfacer de nuevo instanciando las variables con otros hechos del sistema.

Listado 28.47: Primera mejora de la regla atacar

```
1  (defrule EQUIPO-A::atacar1
2     (declare (salience 30))
3     (ficha (equipo "A") (num ?n1) (pos-x ?x1) (pos-y ?y1)
4            (puntos ?p1))
5     (ficha (equipo "B") (num ?n2) (pos-x ?x1) (pos-y ?y2)
6            (puntos ?p2) (descubierta 1))
7     (test (and (> ?y1 ?y2) (> ?p1 ?p2)))
8     (tiempo ?t)
9   =>
10    (assert (mueve (num ?n1) (mov 3) (tiempo ?t)))
11 )
```

Parece evidente que la regla ha mejorado respecto a la aproximación anterior. Pero al comprobar simplemente que la contraria está por arriba existe la posibilidad de suicidio (si entre ellas hay fichas contrarias con más puntos entre ellas), y además se ha restringido mucho la posibilidad de aplicar la regla (ya que si la ficha contraria cambia de columna se dejaría de avanzar hacia ella).

Aunque debido al conocimiento parcial del entorno que tienen los jugadores no existen estrategias netamente mejores unas que otras, lo cierto es que las reglas que son más concretas suelen ser muy efectivas, pero se aplican pocas veces en la partida. Mientras que las reglas más generalistas suelen aplicarse más veces, pero su resultado es menos beneficioso. Por ello se suele buscar un equilibro entre reglas concretas con prioridades altas y otras más generalistas con prioridades más bajas.

Así pues se propone una variante (ojo, que no mejora necesariamente) sobre esta regla para hacerla más aplicable. Simplemente se elimina el condicionante de que estén las dos fichas en la misma columna, resultando la regla del listado 28.48.

Listado 28.48: Segunda mejora de la regla atacar

```
1  (defrule EQUIPO-A::atacar2
2     (declare (salience 20))
3     (ficha (equipo "A") (num ?n1) (pos-y ?y1)
4            (puntos ?p1))
5     (ficha (equipo "B") (num ?n2) (pos-y ?y2)
6            (puntos ?p2) (descubierta 1))
7     (test (and (> ?p1 ?p2) (> ?y1 ?y2)))
8     (tiempo ?t)
9   =>
```

```
10     (assert (mueve (num ?n1) (mov 3) (tiempo ?t)))
11 )
```

Para evitar el suicidio hay que el elemento condicional *not*, que comprueba la no existencia de hechos. Si se desea indicar la no existencia de hechos con determinados valores constantes en sus campos simplemente se antepone *not* a la condición. Por ejemplo *(not (ficha (equipo "A") (num 111)))*. Pero si se quieren incluir comprobaciones más complejas hay que incluir la conectiva "Y", *&*. A continuación se muestra un ejemplo en que se aplicará la una regla de huida del listado 28.49, quedando como ejercicio la aplicación a las reglas anteriores de ataque.

Listado 28.49: Versión inicial de la regla huir

```
1  (defrule EQUIPO-A::huir
2     (declare (salience 20))
3     (ficha (equipo "A") (num ?n1) (pos-x ?x) (pos-y ?y1)
4            (puntos 1))
5     (ficha (equipo "B")           (pos-x ?x) (pos-y ?y2)
6            (puntos 5))
7     (test (> ?y1 ?y2))
8
9     (tiempo ?t)
10  =>
11    (assert (mueve (num ?n1) (mov 4) (tiempo ?t)))
12
13 )
```

Esta regla, sin embargo, adolece los problemas anteriormente comentados: posibilidad de suicidio y aplicación incluso con fichas contrarias muy lejanas (véase estado de la figura 28.107). Se propone como mejora la regla del listado 28.50.

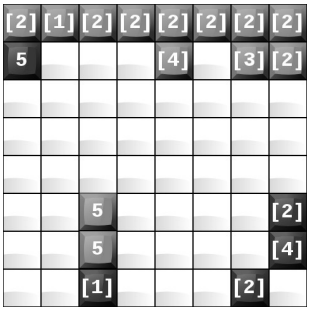


Figura 28.107: Estado de partida para huida.

Listado 28.50: Versión final de la regla huir

```
1  (defrule EQUIPO-A::huir5
2     (declare (salience 20))
3     (ficha (equipo "A") (num ?n1) (pos-x ?x) (pos-y ?y1)
4            (puntos 1))
5     (ficha (equipo "B")           (pos-x ?x) (pos-y ?y2)
6            (puntos 5))
7     (test (or (= ?y1 (+ ?y2 1)) (= ?y1 (- ?y2 1))))
8     (not (ficha (equipo "B") (pos-x ?x1) (pos-y (+ 1 ?y1))))
9     (not (ficha (equipo "B") (pos-x ?x1) (pos-y (- 1 ?y1))))
10    (tiempo ?t)
11  =>
12    (assert (mueve (num ?n1) (mov 1) (tiempo ?t)))
13    (assert (mueve (num ?n1) (mov 2) (tiempo ?t)))
14 )
```

En esta regla se observa otro aspecto muy importante: el no determinismo en la acción. En este caso se proponen dos posible escapatorias, pues se considera que las dos posibilidades son igual de interesantes. De este modo, además, se enriquece la experiencia del usuario, al que le resultará más complicado adivinar el comportamiento implementado en el juego[11]

El no determinismo también se puede (y por general se debe) implementar a nivel de reglas: se pueden tener varias reglas con la misma prioridad que compartan ciertas condiciones, o que las condiciones de una regla sean un subconjunto de las condiciones de otra e incluso condiciones exactamente iguales. Sin embargo,

[11]Nótese que es un ejemplo un poco forzado para mostrar el no determinismo. Realmente sería mejor tener dos reglas, una con una huida para cada lado, que al tener menos condiciones sería más aplicables.

hay veces que los programadores se plantean poner reglas que tengan condiciones excluyentes u ordenar todas las reglas por prioridades. Aunque esta estrategia no es necesariamente mala, se pierde el potencial de los sistemas expertos basados en reglas, pues realmente se estaría implementando un árbol de decisión determinista.

Otro aspecto interesante es que también se pueden crear otros hechos para facilitar la planificación de estrategias. Por ejemplo, si una estrategia cambia su comportamiento cuando pierde las fichas de 5 y 6 puntos se añadiría la primera regla del listado 28.51 y el hecho *fase 2* se usaría como condición de las reglas exclusivas de dicho comportamiento.

Listado 28.51: Regla de cambio de fase

```
(defrule EQUIPO-A::cambioDeFase
   (declare (salience 90))
   (not (ficha (equipo "A") (puntos 5)))
   (not (ficha (equipo "A") (puntos 6)))
 =>
   (assert (fase 2))
)

(defrule EQUIPO-A::despistar
   (declare (salience 50))
   (fase 2)
...
)
```

Hasta aquí llega este capítulo de introducción. Como se habrá observado en los ejemplos una de las ventajas de los sistemas expertos basados en reglas es la facilidad de modificación. El comportamiento de una estrategia se puede cambiar fácilmente añadiendo, modificando o eliminando condiciones de reglas. Igualmente, la re-priorización de reglas es una forma sencilla de refinar estrategias. CLIPS tiene muchas otras funciones y extensiones que se pueden consultar en la documentación en línea del proyecto [75].

El juego que se ha presentado en este capítulo está implementado en *Gades Siege*, disponible bajo licencia libre en su web [99]. El entorno incluye diversas opciones para facilitar su uso, como jugar partidas entre dos estrategias, jugar un humano contra una estrategia, visualizar partidas jugadas o jugar campeonatos. En el web se encuentran instrucciones de instalación y uso, así como un amplio abanico de estrategias de ejemplo resultado de su uso docente en la asignatura *Diseño de Videojuegos* de la Universidad de Cádiz [70].

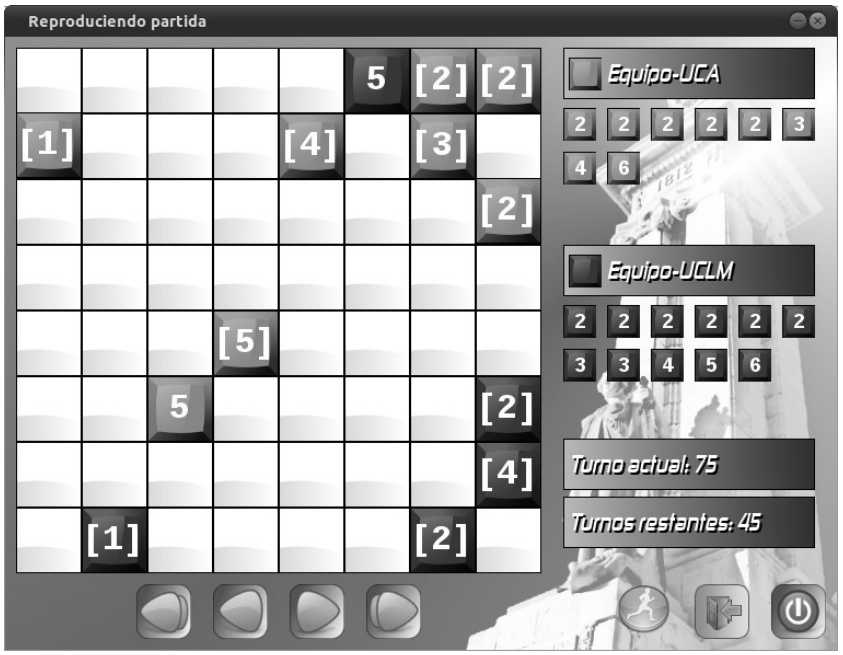


Figura 28.108: Captura de Gades Siege.

Capítulo 29

Networking

Félix Jesús Villanueva
David Villa Alises

Internet, tal y como la conocemos hoy en día, permite el desarrollo de videojuegos en red escalando el concepto de multi-jugador a varias decenas o centenas de jugadores de forma simultánea. Las posibilidades que ofrece la red al mercado de los videojuegos ha posibilitado:

- La aparición de nuevos tipos de videojuegos directamente orientados al juego en red y que suponen, en la actualidad, una gran parte del mercado.
- Evolucionar juegos ya distribuidos mediante actualizaciones, nuevos escenarios, personajes, etc.
- Permitir nuevos modelos de negocio como por ejemplo, distribuyendo el juego de forma gratuita y cobrando por conexiones a los servidores para jugar en red.

29.1. Conceptos básicos de redes

Internet es posible gracias al conjunto de protocolos denominado «pila TCP/IP (Pila de protocolos de Internet)». Un protocolo es un conjunto de reglas sintácticas, semánticas y de temporización que hacen posible la comunicación entre dos procesos cualesquiera.

En todo proceso de comunicaciones están involucrados un conjunto de protocolos cuyo cometido es muy específico y que tradicionalmente se han dividido en capas. Estas capas nos sirven para, conceptualmente, resolver el problema complejo de las comunicaciones aplicando el principio de divide y vencerás.

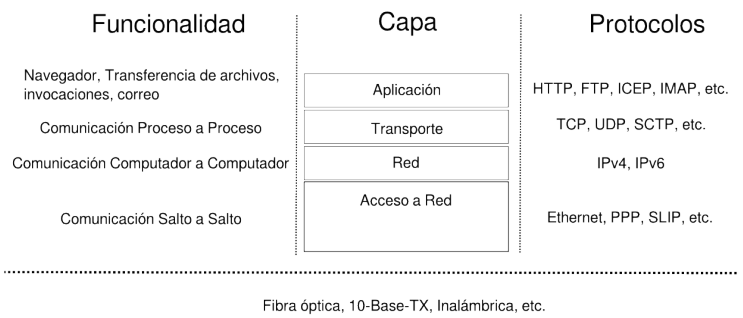


Figura 29.1: Funcionalidades, Capas y Protocolos en Internet

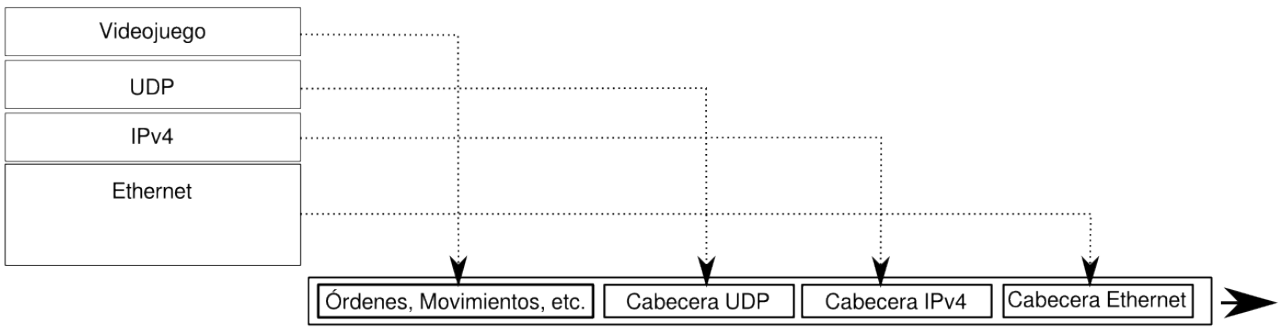


Figura 29.2: Encapsulación de protocolos

Como podemos ver en la figura 29.1 cada funcionalidad necesaria se asocia a una capa y es provista por uno o varios protocolos. Es decir, cada capa tiene la misión específica de resolver un problema y, generalmente, existen varios protocolos que resuelven ese problema de una forma u otra. En función de la forma en la cual resuelve el problema el protocolo, dicha solución tiene unas características u otras. En la capa de aplicación la funcionalidad es determinada por la aplicación.

La agregación de toda esta información dividida en capas se realiza mediante un proceso de encapsulación en el cual los protocolos de las capas superiores se encapsulan dentro de las capas inferiores y al final se manda la trama de información completa. En el *host* destino de dicha trama de información se realiza el proceso inverso. En la imagen 29.2 podemos ver un ejemplo de protocolo desarrollado para un videojuego (capa de aplicación) que utiliza UDP (User Datagram Protocol) (capa de transporte), IPv4 (Capa de red) y finalmente Ethernet (capa de Acceso a Red).

En las siguientes secciones vamos a dar directrices para el diseño e implementación del protocolo relacionado con el videojuego así como aspectos a considerar para seleccionar los protocolos de las capas inferiores.

29.2. Consideraciones de diseño

Desde el punto de vista de la red, los juegos multijugador se desarrollan en tiempo real, esto es, varios jugadores intervienen de forma simultánea, y por tanto son mas exigentes. Su desarrollo y diseño tiene que lidiar con varios problemas:

- Sincronización, generalmente, identificar las acciones y qué está pasando en juegos de tiempo real requiere de una gran eficiencia para proporcionar al usuario una buena experiencia. Se debe diseñar el sistema de transmisión y sincronización así como los turnos de red para que el juego sea capaz de evolucionar sin problemas.
- Identificación de las actualizaciones de información y dispersión de dicha información necesaria a los jugadores específicos para que, en cada momento, todos las partes involucradas tengan la información necesaria del resto de las partes.
- Determinismo que asegure que todas las partes del videojuego son consistentes.

La forma en la que se estructura y desarrolla un videojuego en red, puede determinar su jugabilidad desde el principio del proceso de diseño.

Se debe identificar, desde las primeras fases del desarrollo, qué información se va a distribuir para que todas las partes involucradas tengan la información necesaria para que el juego evolucione.

Al mas bajo nivel, debemos diseñar un protocolo de comunicaciones que, mediante el uso de *sockets* (como veremos en las siguientes secciones), nos permita transmitir toda la información necesaria en el momento oportuno. Como cualquier protocolo, éste debe definir:

Sintaxis Qué información y cómo se estructura la información a transmitir. Esta especificación va a definir la estructura de los mensajes, su longitud, campos que vamos a tener, etc. El resultado de esta fase de diseño debe ser una serie de estructuras a transmitir y recibir a través de un *socket* TCP o UDP.

Semántica Qué significa la información transmitida y cómo interpretarla una vez recibida. Directamente relacionada con la sintaxis, la interpretación de la información de este proceso se realiza mediante el parseo de los mensajes transmitidos e interpretando la información recibida. De igual forma, se construyen los mensajes a transmitir en función de la semántica que queramos transmitir.

Temporización El modelo de temporización expresa la secuencia de mensajes que se deben recibir y transmitir en función de los mensajes recibidos y enviados con anterioridad y de la información que necesitemos o queramos transmitir. La temporización y la semántica generalmente se traducen en una máquina de estados cuyo salto entre estados lo determina los mensajes recibidos y transmitidos y la información contenida en ellas.

El protocolo de un videojuego debe especificar la sintaxis, semántica y temporización

Un aspecto determinante en el diseño es qué información es necesario transmitir. Esta pregunta es específica del videojuego a desarrollar y por lo tanto tiene tantas respuestas como videojuegos en red existen. De forma genérica, debemos identificar:

- Aquella información relacionada con el aspecto visual de un objeto y su estado como puede ser: posición, orientación, movimiento, acción que está realizando, etc.
- Información relativa a la lógica del juego siempre y cuando sea necesario su transmisión en red.
- Aquellos eventos que un jugador puede realizar y que afectan al resto y al propio juego (es decir, al estado del mismo).

La eficiencia en la transmisión y procesamiento de la información de red, así como minimizar la información transmitida, deben regir el diseño y la implementación de la parte de *networking*

Aunque es posible plantear arquitecturas de comunicaciones P2P (Peer To Peer) o con múltiples servidores (como veremos posteriormente), en la mayor parte de los juegos en red, cliente y servidor tienen tareas bien diferenciadas. El cliente envía su posición y representa al enemigo en la posición que recibe. Se encarga también del tratamiento de la información hacia y desde los periféricos del computador (teclado, ratón, jostick, micrófono, etc.) y la envía al servidor. El servidor por su parte mantiene el mundo en el que están representados todos los jugadores, calcula la interacciones entre ellos y envía a los clientes información de forma constante para actualizar su estado.

Podemos ver un ejemplo de esta implementación de *networking* en los motores de juego de VALVE [22] que simulan eventos discretos para su mundo virtual entorno a 33 veces por segundo. El motor del juego del servidor envía actualizaciones de estado unas 20 veces por segundo a los clientes que guardan los 100 últimos milisegundos recibidos. De forma simultánea cada cliente envía el estado del ratón y teclado entorno a 20 veces por segundo (no se genera un evento por cada cambio de estado sino que se discretiza a intervalos periódicos).

Obviamente estos datos pueden modificarse de forma estática en función del tipo de juego que estemos desarrollando (por ejemplo, en los juegos de estrategia suelen ser mas amplios) o de forma dinámica en función de los clientes conectados. Es necesario resaltar que en el caso de clientes conectados con diversas características en conexiones de red, la velocidad se debe ajustar al mas lento (dentro de una calidad aceptable). Estos ajustes dinámicos se suelen hacer mediante pruebas de *ping* que ayudan a ver el retardo completo existente en la red. Habitualmente jugadores con un retardo excesivo son rechazados puesto que degradarían el rendimiento global del juego.

En cualquier caso, cada tipo de juego tiene unos requerimientos aceptables en cuanto a latencia y el estudio del comportamiento de los jugadores nos pueden dar indicaciones en cuanto a frecuencia de envío de comandos, latencia aceptable, etc.

A pesar de la evolución en ancho de banda de las redes actuales, es necesario establecer mecanismos para mejorar la eficiencia de las comunicaciones y dotar a clientes y servidores de mecanismos para proporcionar un rendimiento aceptable aún en presencia de problemas y pérdidas en la red. Este tipo de mecanismos son, principalmente [22]:

Compresión de datos La compresión de datos entendida como las técnicas destinadas a reducir el ancho de banda necesario para el juego multijugador. Entre las técnicas utilizadas podemos citar:

- Actualizaciones incrementales, es decir, enviar a los clientes sólo y exclusivamente la información que ha cambiado con respecto a la última actualización enviada. En esta técnica, de forma general, cada cierto tiempo se envía un estado completo del mundo para corregir posibles pérdidas de paquetes.
- Seleccionar el destino de la información transmitida en función de qué clientes se ven afectados por el cambio de estado de los objetos en lugar de transmitir toda la información a todos.
- agregación: envío de datos comunes a varios jugadores con un solo paquete gracias, por ejemplo, el uso de comunicaciones multicast.

Interpolación La interpolación permite obtener nuevos estados del mundo virtual del juego en el cliente a partir de los datos recibidos del servidor. En función de la complejidad de estos nuevos estados siempre son una aproximación al estado que tendríamos si obtuviéramos eventos de estado del servidor a una altísima frecuencia. Esta técnica nos permite actualizar el estado del juego en el cliente mas a menudo que las actualizaciones enviadas por el SERVIDOR y, al mismo tiempo, poder inferir información perdida en la red.

Predicción La predicción nos ayuda a obtener nuevos estados futuros del mundo virtual del juego en el cliente a partir de los datos recibidos del servidor. La diferencia respecto a la interpolación es que la ésta genera nuevos estados entre dos estados recibidos por el servidor de cara a la renderización mientras que la predicción se adelanta a los eventos recibidos desde el servidor. Para ello, el cliente puede predecir su movimiento en función de la entrada que recibe del usuario y sin que el servidor se lo comunique.

Compensación del retardo A pesar de las técnicas anteriores siempre las acciones cuentan con un retardo que se debe compensar, en la medida de lo posible, guardando los eventos recibidos por los clientes y sincronizando el estado del mundo de forma acorde a como estaba el mundo cuando el evento fue originado, es decir, eliminando el intervalo de comunicación.

Una vez implementadas estas técnicas se debe estimar y medir los parámetros de *networking* para ver la efectividad de las técnicas, estudiar el comportamiento de la arquitectura, detectar cuellos de botella, etc... Por lo tanto, las estimaciones de retardo y en general, todos los parámetros de red deben ser incluidos en los tests.

Con estas técnicas en mente, la responsabilidad del cliente, a grandes rasgos, queda en un bucle infinito mientras se encuentre conectado. En dicho bucle, para cada turno o intervalo de sincronización, envía comandos de usuario al servidor, comprueba si ha recibido mensajes del servidor y actualiza su visión del estado del juego si es así, establece los cálculos necesarios: predicción, interpolación, etc. y por último muestra los efectos al jugador: gráficos, mapas, audio, etc.

El servidor por su parte, actualiza el estado del mundo con cada comando recibido desde los clientes y envía las actualizaciones a cada uno de los clientes con los cambios que se perciben en el estado del juego.

En ambos casos se necesita un algoritmo de sincronización que permita a clientes y servidores tener consciencia del paso del tiempo. Existen varios algoritmos y protocolos empleados para esta sincronización (a menudo heredados del diseño y desarrollo de sistemas distribuidos).

Un algoritmo básico espera a obtener comandos de todos los clientes involucrados en el juego, realiza los cómputos necesarios y a continuación salta al siguiente *slot* de sincronización donde, de nuevo, espera todos los comandos de los clientes.

Otro de los algoritmos básicos consta de una espera fija que procesa sólo los comandos recibidos hasta ese momento, a continuación, pasa al siguiente slot.

En cualquier caso, siempre existe un retardo en las comunicaciones por lo que los comandos de los usuarios llegan con retardo al servidor. De cara a compensar este retardo y que no influya en el juego se suelen realizar compensaciones de retardo que, junto con estimaciones locales de las consecuencias de los comandos de usuario, permiten mitigar completamente este retardo de la experiencia del jugador.

29.3. Eficiencia y limitaciones de la red

Diseñar e implementar un videojuego implica realizar un uso eficiente de los recursos del computador, que atañe principalmente a la capacidad de cómputo, la memoria, la representación gráfica, sonido e interfaces de interacción física con el usuario.

Si además el juego requiere que sus distintos componentes se ejecuten en varios computadores conectados en red, el diseño y la propia arquitectura de la aplicación puede cambiar drásticamente. Existen tres modelos de comunicación básicos que se utilizan en los «juegos en red».

29.3.1. Peer to peer

Varias instancias idénticas del programa colaboran intercambiándose información. El programa que ejecuta el usuario tiene toda la lógica y el modelo del juego y puede funcionar de forma completamente autónoma. Era una técnica muy utilizada en los juegos de los 90, especialmente en los RTS y los juegos sobre LAN (Local Area Network)[1], como por ejemplo Starcraft, de Blizzard®.

Cada instancia del juego comparte información con todas las demás. El modelo del juego suele corresponder con una secuencia de *turnos* (denominado *peer-to-peer lockstep*) al comienzo de los cuales se produce un intercambio de mensajes que define el *estado* del juego. En principio, esto garantiza un estado global determinista, pero en la práctica supone problemas graves:

- Dado que supone un aumento exponencial del número de mensajes (todos hablan con todos) es un modelo que escala mal para cantidades de pocas decenas de usuarios, si no se está utilizando una LAN.
- Además, como se requiere el *visto bueno* de todos los *pares*, la duración de cada turno estará determinada por el *par* con mayor latencia.
- Como existen n instancias concurrentes del juego que evolucionan de forma independiente, es relativamente fácil que haya discrepancias en la evolución del juego en distintas máquinas resultando en situaciones de juego diferentes a partir de un estado inicial supuestamente idéntico.
- Añadir una nueva instancia a un juego en marcha suele ser problemático puesto que puede resultar complicado identificar y transmitir un estado completamente determinista al nuevo jugador.

[1] Los juegos de esta época utilizaban normalmente el protocolo IPX para las comunicaciones en LAN

Además, el modelo *peer-to-peer* es vulnerable a varias técnicas de *cheating*:

- Latencia falsa (*fake latency* o *artificial lag*). Consiste en crear problemas de conectividad o carga en la red hacia la instancia que utiliza el jugador tramposo (*cheater*) para alterar el funcionamiento de todas las otras instancias. Como todos los jugadores han de esperar los comandos de estado de los demás jugadores, el resultado es una grave pérdida de fluidez dejando el juego aparentemente bloqueado al final de cada turno. El *tramposo* puede aprovechar esta situación para dificultar a otros que su personaje sea localizado o alcanzado por disparos en un *shooter*. Se puede implementar sin necesidad de acceder al código fuente del juego, en ocasiones con un método tan rudimentario como un pulsador físico en el cable de red.
- Look-ahead. Con esta técnica el tramposo retrasa sus mensajes de estado a los otros pares hasta que conoce la información que le interesa: posición y estado de los oponentes. Después crea un mensaje de estado con una marca de tiempo en el pasado, que corresponde al momento en el que debió enviarla realmente. Las otras instancias del juego interpretan que el mensaje llegó tarde debido a la latencia y puede ser difícil de detectar que se trata de un mensaje sintético.

29.3.2. Cliente-servidor

Con la llegada de los primeros FPS y la adopción masiva de internet (mediante módem) a mediados de los 90, los efectos de la latencia de los que adolece especialmente el modelo *peer-to-peer* obligaron a los desarrolladores a buscar otras alternativas. Uno de los primeros juegos en probar algo diferente fue Quake, de «id Software®»

En el modelo cliente-servidor muchos jugadores se conectan a un único servidor que controla el juego y mantiene la imagen global de lo que ocurre. Este modelo elimina muchos de los problemas y restricciones del modelo *peer-to-peer*: ya no se requiere mantener una realidad coherente compartida por todos los jugadores, puesto que únicamente el servidor determina el estado del juego. Los clientes muestran dicho estado con pequeñas modificaciones que representan la evolución inmediata del escenario en función de la entrada del usuario. La topología lógica pasó de ser una malla completa a una estrella de modo que mejoró la escalabilidad y redujo la latencia; ya no era necesario esperar al cliente más lento. También simplifica la incorporación de nuevos jugadores a una partida en curso.

Este modelo desacopla en muchos casos el desarrollo del juego y las interacciones entre los jugadores simplificando las tareas de la parte cliente.

29.3.3. Pool de servidores

Cuando la cantidad de jugadores es muy grande (cientos o miles) se utilizan varios servidores coordinados de modo que se pueda balancear la carga. Sin embargo, su sincronización puede ser compleja. Se podría ver como un modelo híbrido que incorpora características de los dos anteriores. Sin embargo, no adolece de los problemas que el modelo *peer-to-peer* (o al menos no a la misma escala) puesto que los servidores que forman el *pool* se ejecutan en computadores de alto rendimiento, con conexiones de alto ancho de banda y bajo el control de la compañía y su número es mucho menor que uno por usuario.

29.4. Restricciones especificas de los juegos en red

Cuando la red entra en juego debemos tener en cuenta tres limitaciones adicionales además de las habituales:

29.4.1. Capacidad de cómputo

Aunque la utilización de la CPU ya es por sí misma un factor determinante en la mayoría de los videojuegos modernos, en el caso de un juego en red debemos tener en cuenta además la sobrecarga de cómputo que implica el procesamiento y generación de los mensajes, aparte de las relativamente costosas técnicas de compensación de la latencia que veremos a continuación.

29.4.2. Ancho de banda

Es la tasa de transferencia efectiva (en bits por segundo) entre los distintos computadores que alojan partes del juego. Si existen flujos asimétricos deben considerarse por separado. Es también habitual que entre dos componentes existan flujos de control y datos diferenciados. Resulta determinante cuando se deben soportar grandes cantidades de jugadores y es el aspecto que más limita la escalabilidad de la arquitectura cliente-servidor.

29.4.3. Latencia

Es el tiempo que tarda un mensaje desde que sale del nodo origen hasta que llega al nodo destino. Hay muchos factores que afectan a la latencia: las limitaciones físicas del medio, los procedimientos de serialización de los mensajes, el sistema operativo, el tiempo que los paquetes pasan en las colas de los encaminadores, los dispositivos de conmutación, etc.

Muy relacionado con la latencia está el *tiempo de respuesta*, que es el tiempo total desde que se envía un mensaje hasta que se recibe la respuesta. En ese caso se incluye también el tiempo de procesamiento del mensaje en el servidor, además de la latencia de ida y vuelta. Un juego con restricciones importantes debería evitar depender de información que se obtiene como respuesta a una petición directa. Es frecuente utilizar mensajes con semántica *oneway*, es decir, que no implican esperar una respuesta en ese punto.

Aún hay otra consideración muy importante relacionada con la latencia: el *jitter*. El jitter es la variación de la latencia a lo largo del tiempo. Cuando se utilizan redes basadas en conmutación de paquetes con protocolos *best-effort* —como IP— las condiciones puntuales de carga de la red en cada momento (principalmente los encaminadores) afectan decisivamente al retardo de modo que puede cambiar significativamente entre un mensaje y el siguiente.

La latencia y el jitter permisible dependen mucho del tipo de juego y están muy influenciados por factores psicomotrices, de percepción visual del espacio, etc. Los juegos más exigentes son los que requieren control continuo, como la mayoría de los *shotters*, simuladores de vuelo, carreras de coches, etc., que idealmente deberían tener latencias menores a 100 ms. El punto de vista subjetivo aumenta la sensación de realismo y provoca que el jugador sea mucho más sensible, afectando a la jugabilidad. En otros tipos de juegos como los de estrategia o que dependen de decisiones colaborativas, el retardo admisible puede ser de hasta 500 ms o incluso mayor.

La evolución de los modelos de comunicaciones en los videojuegos en red y la mayoría de las técnicas que veremos a continuación tienen un único objetivo: compensar los efectos que la latencia tienen en la experiencia del usuario.

29.5. Distribución de información

Existen muchas técnicas para reducir el impacto de las limitaciones del ancho de banda y la latencia en las comunicaciones. Por ejemplo, para reducir el ancho de banda necesario se pueden comprimir los mensajes, aunque eso aumenta la latencia y el consumo de CPU tanto en el emisor como en el receptor.

También se puede mejorar la eficiencia agrupando mensajes (*message batching*). La mayor parte de los mensajes que se intercambian los componentes de un juego son bastante pequeños (apenas unos cientos de bytes). Utilizar mensajes mayores (cercanos a la MTU (Maximum Transfer Unit)) normalmente es más eficiente, aunque de nuevo eso puede afectar negativamente a la latencia. El primer mensaje de cada bloque tendrá que esperar a que la cantidad de información sea suficiente antes de salir del computador. Obviamente eso puede ser prohibitivo dependiendo de la información de que se trate. La posición del jugador debería notificarse lo antes posible, mientras que su puntuación actual puede esperar varios segundos.

Como en tantas otras situaciones, las técnicas que permiten mejorar un aspecto a menudo son perjudiciales para otros. Por ese motivo no se pueden dar pautas generales que sean aplicables en todos los casos. En cada juego, y para cada flujo de información habrán de analizarse las opciones disponibles.

Una de las técnicas más útiles consiste en modelar el estado de los objetos —en particular los jugadores y el mundo virtual— de modo que los cambios puedan ser notificados de forma asíncrona, es decir, limitar al mínimo las situaciones en las que la progresión del juego dependa de una respuesta inmediata a una petición concreta o de un evento asíncrono (como el protocolo *lockstep*). Como se ha visto en otros capítulos, la mayoría de los juegos se pueden modelar como un sistema dirigido por eventos y no es diferente en el caso de la interacción con componentes remotos.

Una petición asíncrona es aquella que no bloquea al programa, habitualmente al cliente. Eso se puede conseguir realizando la petición en un hilo de ejecución adicional y especificando un *callback*, permitiendo así progresar al juego. Cuando llega la respuesta, se trata como un evento asíncrono más.

Otra posibilidad interesante —siguiendo la misma idea— es utilizar comunicaciones *oneway*, es decir, enviar mensajes que anuncian cambios (hacia o desde el servidor) pero que no implican una respuesta de vuelta a la fuente del mensaje. Los mensajes *oneway* (que podemos llamar *eventos*) permiten diseñar estrategias mucho más escalables. Una comunicación basada en intercambio de mensajes petición–respuesta solo se puede implementar con un sistema de entrega *unicast* (un único destino por mensaje) mientras que una comunicación basada en eventos puede aprovechar los mecanismos de difusión multicast de TCP/IP y otros protocolos.

Es bien sabido que Internet tiene un soporte de multicast bastante limitado debido a que hay relativamente pocas operadoras que ofrezcan ese servicio de forma generalizada. Además, IP multicast solo es aplicable si se utiliza UDP como protocolo de transporte (aunque es los más frecuente en los juegos en red). Incluso con estas restricciones, es posible utilizar mecanismos de propagación de eventos que permiten que las comunicaciones del juego resulten mucho más eficientes y escalables. En juegos con miles de jugadores y decenas de servidores, la comunicación basada en eventos simplifica la sincronización de estos y los algoritmos de consistencia.

29.6. Modelo de información

En un juego en red hay mucha información estática (como el mapa del nivel en un *shotter*), pero también muchos datos que se actualizan constantemente. La responsabilidad de almacenar, actualizar y transmitir toda esa información lleva a implementar diferentes modelos, que obviamente están muy relacionados con los modelos de comunicaciones que vimos en la sección anterior:

Centralizado Es el modelo más sencillo. Un único componente (el servidor) recoge la información procedente de los jugadores, actualiza el estado global del juego, comprueba que es completo y correcto aplicando las reglas del juego y otras restricciones y distribuye el resultado a los jugadores. Esto incluye evaluar la interacción entre los jugadores, detectar errores o trucos ilegales (*cheats*) o actualizar las puntuaciones. Habrá información que interese a todos los jugadores por igual y deba ser transmitida a todos ellos, pero habrá ciertos datos que pueden implicar únicamente a algunos de los jugadores. Por ejemplo, la orientación y ángulo exacto del arma de cada jugador solo es importante para aquellos que pueden verlo o ser alcanzados si dispara. Este tipo de análisis puede ahorrar muchos mensajes innecesarios. En resumen, en el modelo centralizado únicamente el servidor conoce el mundo completo y cada jugador solo tiene la información que le atañe.

Replicado En este caso, varios (incluso todos) los participantes en el juego tienen una copia del mundo virtual. Eso implica la necesidad de sincronizar esa información entre los participantes. Por eso, resulta conveniente cuando la mayor parte del estado del juego puede ser determinada de forma aislada, es decir, con poca interacción con los participantes.

Distribuido Cada participante tiene una parte del estado global del juego, normalmente la que más le afecta, de modo que se reducen los mensajes necesarios para determinar la situación de cada jugador. El modelo distribuido es adecuado cuando el estado depende mucho de variables difíciles de prever. Lógicamente, es menos adecuado para evitar inconsistencias.

29.7. Uso de recursos de red

El uso que cualquier aplicación hace de los recursos de red se puede cuantificar en función de ciertas variables objetivas [90]:

- El número de mensajes transmitidos.
- La cantidad media de destinatarios por mensaje.
- El ancho de banda medio requerido para enviar un mensaje a cada destinatario.
- La urgencia de que el mensaje llegue a cada destinatario.
- La cantidad de procesamiento necesario para procesar cada mensaje entrante.

Estos valores pueden ser medidos en una situación de funcionamiento óptimo o bien pueden ser calculados a partir de un modelo teórico del juego. También es posible determinar cómo afectarán a estas variables los posibles cambios respecto a una situación de partida. Por ejemplo, en qué medida crece la cantidad o el tamaño de los mensajes en función del número de jugadores, el tamaño del mapa, la latencia del jugador más lento, etc.

29.8. Consistencia e inmediatez

La consistencia se puede cuantificar en función de las diferencias entre el estado del juego que percibe cada jugador respecto al estado de referencia que mantiene el servidor. Estas diferencias se deben a la dificultad de compartir la misma información con todos los nodos en el mismo instante. Existen algoritmos distribuidos que permiten obtener un estado global consistente, por ejemplo utilizando reglas de orden causal. Sin embargo, la obtención de ese estado global lleva demasiado tiempo, aumentando sensiblemente la latencia. Por eso muchas veces es preferible sacrificar algo de consistencia en favor de la inmediatez.

La inmediatez (del inglés *responsiveness*) es el tiempo que necesita un participante para procesar un evento que afecta al estado global. Si cada nodo procesa cada evento tan pronto como le llega, el jugador tendrá realimentación inmediata de sus acciones y las de otros, pero si lo hace sin verificar que el evento ha llegado también a los demás participantes se producirán inconsistencias. Es decir, consistencia e inmediatez son propiedades esenciales, pero contrapuestas.

Cuanto menor sea el ancho de banda, mayor la latencia y el número de participantes, más difícil será mantener un equilibrio razonable entre consistencia e inmediatez.

Desde el punto de vista del jugador es determinante que el juego reaccione inmediatamente, al menos a las acciones que realiza él. En el modelo cliente-servidor puro las acciones del jugador (ej. pulsaciones de teclas) se envían al servidor y la representación de sus acciones en su interfaz gráfica no ocurre hasta que el servidor las valida y determina la posición y estado del jugador. Esto supone entre 100 y 200 ms en el mejor de los casos si intervienen conexiones WAN. Algunos juegos cliente-servidor utilizan un pequeño «truco» que se aprovecha de la percepción humana. Si el sistema sensorial humano percibe un estímulo visual y otro sonoro que se supone que son simultáneos (ej. una puerta que se cierra y el sonido que produce) el cerebro lo interpreta así a pesar de que no se produzcan realmente en el mismo instante. La tolerancia máxima que parece aplicar el cerebro ronda los 100-150 ms. Este efecto se denomina «vinculación multisensorial».

vinculación multisensorial

El jugador dispara su arma, el cliente reproduce el sonido del disparo inmediatamente y envía un mensaje al servidor. El servidor envía un mensaje indicando la rotura de un cristal como efecto del disparo. Si todo ello ha ocurrido en un lapso de alrededor de 100 ms, el cerebro del jugador interpretará que todo ha ocurrido simultáneamente.

Una mejora obvia es conseguir que el cliente calcule los cambios en el estado que estima corresponden a la interacción con el jugador previendo lo que más tarde anunciará el servidor. Obviamente, el servidor dispone de la información de todos los participantes mientras que el participante solo dispone de la interacción con el jugador y la historia pasada.

Veamos un ejemplo. El jugador presiona el botón de avance, la aplicación calcula la nueva posición del personaje y la representa. Al mismo tiempo un mensaje que incluye esa acción se ha enviado al servidor. El servidor realiza el mismo cálculo y envía a todos los clientes involucrados la nueva posición del personaje. En condiciones normales, habrá relativamente poca variación entre la posición calculada por el cliente y por el servidor y no será necesario que el cliente tome ninguna acción adicional.

Pero puede presentarse otra situación. En el preciso momento en el que el jugador avanzaba el personaje recibe un disparo. El servidor descarta el mensaje con la acción de avance y notifica la muerte de ese personaje. Todos los participantes han visto morir al personaje mientras que el jugador ha sido testigo de la inconsistencia, primero lo vio avanzar normalmente y luego de pronto aparecer muerto unos metros atrás.

Eco local

Un ejemplo cotidiano de la relación entre consistencia e inmediatez lo tenemos en las aplicaciones de shell o escritorio remoto. En estas aplicaciones suele existir la opción de utilizar *eco local* o *eco remoto*, es decir, si la aparición de las pulsaciones o posición del ratón depende directamente de nuestras acciones o de la realimentación que dé el servidor.

Es una mejora obvia pero compleja de implementar. Para que el cliente pueda determinar el estado previsible de personaje necesita aplicar todas las restricciones físicas del escenario. En el caso de un juego 3D eso implica colisiones, gravedad, inercia, ..., cosas que no eran necesarias con un modelo cliente-servidor puro.

Un modelo altamente consistente no puede hacer este tipo de predicciones, pero tendrá poca inmediatez (es decir alta latencia) porque las acciones deben ser corroboradas por el servidor antes de ser consideradas válidas. Si en lugar de un modelo centralizado utilizamos un modelo distribuido la situación es más grave porque la acción deberá ser corroborada por todos los implicados.

29.9. Predicciones y extrapolación

Cuando la inmediatez es un requisito más importante que la consistencia, la solución pasa por predecir. Para conseguir mayor inmediatez se requiere mínima latencia, y dado que se trata de una restricción física la única alternativa es predecir. A partir de los valores previos de las propiedades de un objeto, se extrapola el siguiente valor probable. De ese modo, el usuario percibirá un tiempo de respuesta cercano a cero, similar al de un juego monojugador. En la sección anterior vimos un esbozo para conseguir predecir la consecuencias de las acciones del usuario (esto se denomina «predicción del jugador»), pero también podemos predecir las acciones de las otras partes activas del juego, que se denomina en general «predicción del oponente» o «del entorno» [10].

29.9.1. Predicción del jugador

Veamos en detalle el algoritmo que aplica el cliente para predecir cómo las acciones del usuario afectan al estado del juego:

1. Procesar eventos procedentes de los controles (teclado, ratón, joystick, etc.).
2. Serializar las acciones del usuario y enviarlas al servidor.
3. Determinar qué elementos del mundo virtual serán influidos con la acción del usuario y calcular las modificaciones en su estado.
4. Renderizar la nueva situación del juego.
5. Esperar las actualizaciones de estado del servidor.
6. Si hay diferencias, modificar el estado del juego para compensar los errores cometidos.

Aunque el usuario tiene realimentación inmediata de sus acciones, el tiempo transcurrido entre el paso 4 y el 6 (el tiempo de respuesta) sigue siendo determinante. Si las predicciones no son rápidamente corroboradas o corregidas con la información procedente del servidor, el error cometido por la predicción se irá acumulando aumentando las diferencia entre el estado real del juego (mantenido por el servidor) y el estimado por el cliente.

Nótese que el cliente puede además enviar la predicción al servidor, de modo que éste pueda validarla. Así el servidor solo enviará datos si detecta que el error ha superado determinado umbral. Mientras el cliente sea capaz de predecir correctamente la trayectoria, el servidor simplemente tiene que dar el visto bueno, y no tiene porqué hacerlo constantemente. Esto reduce el tráfico desde el servidor hacia los clientes, pero le supone una carga de trabajo adicional.

El problema surge cuando el servidor envía correcciones respecto a la predicción. La corrección tendrá una marca de tiempo en el pasado, de modo que el cliente debería desechar todos las modificaciones que ha hecho sobre el estado el juego desde la marca hasta el presente. Después debe volver a aplicar todos los eventos que se han producido desde ese instante para estimar el nuevo estado.

Eso significa que el estado previsto por el cliente que no han sido corroborada por el servidor debe considerarse tentativo, y los eventos que se han producido desde el último instante verificado deben almacenarse temporalmente.

29.9.2. Predicción del oponente

Esta técnica (también conocida como *dead reckoning* [10, § 9.3]) se utiliza para calcular el estado de entidades bajo el control de otros jugadores o del motor del juego.

Para predecir la posición futura de un objeto se asume que las características de su movimiento (dirección y velocidad) se mantendrán durante el siguiente ciclo. Como en la predicción del jugador, el cliente puede publicar sus predicciones. El servidor (o los otros participantes si es una arquitectura distribuida) verifica el error cometido y decide si debe enviar la información real para realizar las correcciones. Esta información puede incluir la posición, dirección y sentido, velocidad e incluso aceleración. De ese modo, el cliente no necesita estimar o medir esos parámetros, puede utilizar los que proporciona la fuente, sea otro jugador o el propio motor del juego, ahorrando cálculos innecesarios. La figura 29.3 ilustra este proceso.

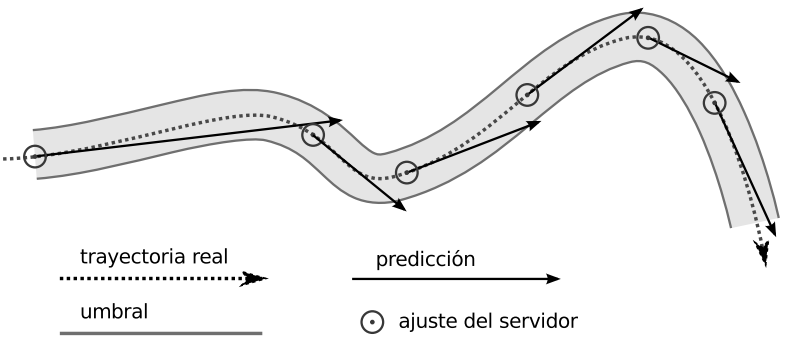


Figura 29.3: Cuando la predicción calcula una posición fuera del umbral permitido (fuera de la línea roja), el servidor envía una actualización (marcas azules) que incluye la posición, dirección y velocidad real actual. El cliente recalcula su predicción con la nueva información

Conocido el vector de dirección y la velocidad es sencillo predecir la operación con física elemental (ver formula 29.1), siendo t_0 el instante en que se calculó la posición anterior y t_1 el instante que se desea predecir.

$$e(t_1) = e(t_0) + v(t_1 - t_0) \tag{29.1}$$

Si además se desea considerar la aceleración se utiliza la formula para un movimiento uniformemente acelerado y rectilíneo:

$$e(t_1) = e(t_0) + v(t_1 - t_0) + \frac{a(t_1 - t_1)^2}{2} \tag{29.2}$$

Considerar la aceleración puede llevar a cometer errores de predicción mucho mayores si no se tiene realimentación rápida, por lo que se desaconseja con latencias altas. El cliente puede medir continuamente el RTT (Round Trip Time) al servidor y en función de su valor decidir si es conveniente utilizar la información de aceleración.

Dependiendo del tipo de juego, puede ser necesario predecir la posición de partes móviles, rotaciones o giros que pueden aumentar la complejidad, pero no hacerlo podría llevar a situaciones anómalas, como por ejemplo un personaje que debe apuntar continuamente a un objetivo fijado.

Cuando la diferencia entre el estado predicho y el real (calculado por el servidor) es significativa no es admisible modificar sin más el estado del juego (*snap*). Si se trata, por ejemplo, de la posición de un personaje implicaría saltos (se *teletransporta*), que afectan muy negativamente a la experiencia de juego. En ese caso se utiliza un algoritmo de *convergencia* que determina los ajustes necesarios para llegar a un estado coherente con el estado real desde la posición incorrecta en el pasado en un tiempo acotado. Si se trata de posiciones se habla de «trayectoria de convergencia lineal». La figura 29.4 ilustra la diferencia entre aplicar o no un algoritmo de convergencia de trayectoria para el ejemplo anterior.

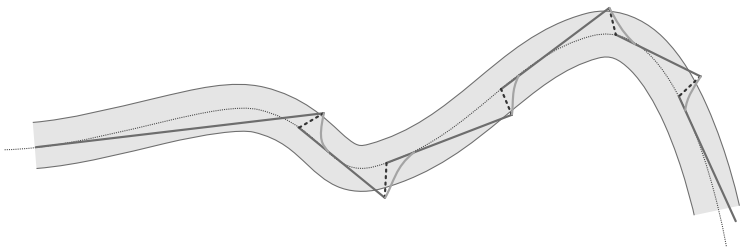


Figura 29.4: Cuando se reajusta la posición se producen saltos (líneas discontinuas azules) mientras que aplicando un algoritmo de convergencia (líneas verdes) el objeto sigue una trayectoria continua a pesar de no seguir fielmente la trayectoria real y tener que hacer ajustes de velocidad

Aunque es una técnica muy efectiva para evitar que el usuario sea consciente de los errores cometidos por el algoritmo de predicción, pueden presentarse situaciones más complicadas. Si en un juego de carreras, la ruta predicha implica que un coche sigue en línea recta, pero en la real toma un desvío, la ruta de convergencia llevaría el coche a atravesar una zona sin asfaltar, que puede transgredir las propias reglas del juego. Este escenario es muy común en los receptores de GPS (Global Positioning System) que utilizan una técnica de predicción muy similar (ver figura 29.5).

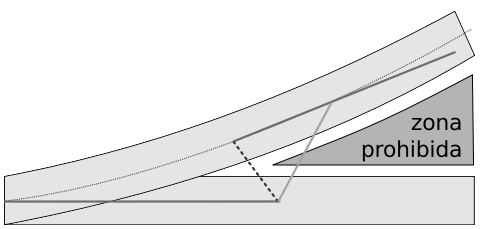


Figura 29.5: Ejemplo de trayectoria de convergencia incorrecta debido a un obstáculo

29.10. Sockets TCP/IP

La programación con *sockets* es la programación en red de mas bajo nivel que un desarrollador de videojuegos generalmente realizará. Por encima de los *sockets* básicos que veremos a lo largo de esta sección se pueden usar librerías y middlewares, algunos de los cuales los veremos mas adelante.

Estas librerías y middlewares nos aportan un nivel mas alto de abstracción aumentando nuestra productividad y la calidad del código, a costa, en algunas ocasiones, de una pérdida de la eficiencia.

Conceptualmente, un *socket* es un punto de comunicación con un programa que nos permite comunicarnos con él utilizando, en función del tipo de *socket* que creemos, una serie de protocolos y, a más alto nivel, mediante un protocolo que nosotros mismo diseñamos. En función del rol que asuma el programa, cabe distinguir dos tipos de programas:

- Cliente: Que solicita a un servidor un servicio.
- Servidor: Que atiende peticiones de los clientes.

En la literatura tradicional, hablamos de mecanismos de IPC (InterProcess Communication) (*Inter-Process Communication*) a cualquier mecanismo de comunicación entre procesos. Cuando los procesos a comunicar se encuentran dentro del mismo computador, hablamos de programación concurrente. Cuando hay una red de ordenadores entre los dos procesos a comunicar, hablamos de programación en red.

En ambos casos se pueden utilizar *sockets*, no obstante, nosotros vamos a tratar con un tipo específico de *sockets* denominados *Internet Sockets* y que nos permiten comunicar un proceso en nuestra máquina con cualquier otra máquina conectada a Internet.

29.10.1. Creación de *Sockets*

Aunque, por norma general, los *sockets* desarrollados en los sistemas operativos no son exactamente iguales, es cierto que casi todos los sistemas operativos han seguido los denominados *Berkeley Sockets* como guía de diseño y por lo tanto, como API para la programación de aplicaciones en red. En esta sección vamos a seguir los pasos de programación de un entorno GNU/Linux que es extensible a toda la familia UNIX. Con matices mas amplios sirve para entender el interfaz de programación *socket* en general.

Veamos paso a paso la creación de *sockets*, si vemos la primitiva para crear un *socket*:

sockfd = socket(int socket_family, int socket_type, int protocol);

Admite tres argumentos de entrada, el *socket_family* define la familia del *socket* que vamos a crear, aunque en esta sección nos vamos a centrar en la familia AF_INET relacionada con los *sockets* para comunicaciones entre procesos a través de Internet (usando IPv4), es necesario resaltar que se pueden usar *sockets* para cualquier tipo de comunicación inter-proceso. El parámetro que devuelve es un descriptor del *socket* que utilizaremos para referirnos a él.

Siguiendo la filosofía de diseño de los sistemas operativos Unix y Linux, los descriptores de *sockets* son también descriptores de archivos

En concreto, tenemos las familias de *sockets* AF_UNIX y AF_LOCAL para la comunicación local donde la dirección de comunicación es el nombre de un archivo, AF_INET6 para IPv6, AF_NETLINK para comunicar el kernel con el espacio de usuario o procesos entre ellos (en este caso similar a la familia AF_UNIX), etc.

El segundo argumento nos especifica qué tipo de *socket* vamos a crear dentro de la familia seleccionada en el argumento anterior. Para AF_INET *socket* tenemos como principales opciones:

- SOCK_STREAM : indica un tipo de *socket* que soporta una comunicación entre procesos confiable, orientada a la conexión y que trata la comunicación como un flujo de bytes.
- SOCK_DGRAM : indica un tipo de *socket* que soporta una comunicación entre procesos no confiable, no orientada a la conexión y que trata la comunicación como un intercambio de mensajes.

Dentro de esta familia también se incluyen otro tipo de *sockets* como SOCK_RAW que permite al programador acceder a las funciones de la capa de red directamente y poder, por ejemplo, construirnos nuestro propio protocolo de transporte. El otro tipo de *socket* que nos podemos encontrar es SOCK_SEQPACKET que proporciona una comunicación fiable, orientada a conexión y a mensajes. Mientras que las tres familias anteriores están soportadas por la mayoría de los sistemas operativos modernos, la familia SOCK_SEQPACKET no tiene tanta aceptación y es mas difícil encontrar una implementación para según qué sistemas operativos.

Si nos fijamos, los tipos de *sockets* nos describen qué tipo de comunicación proveen. El protocolo utilizado para proporcionar dicho servicio debe ser especificado en el tercer parámetro. Generalmente, se asocian SOCK_DGRAM al protocolo UDP, SOCK_STREAM al protocolo TCP, SOC_RAW directamente sobre IP. Estos son los protocolos que se usan por defecto, no obstante, podemos especificar otro tipos de protocolos siempre y cuando estén soportados por nuestro sistema operativo.

Como ya hemos comentado, cada tipología de *socket* nos proporciona una comunicación entre procesos con unas determinadas características y usa unos protocolos capaces de proporcionar, como mínimo, esas características. Normalmente, esas características requieren de un procesamiento que será mas o menos eficiente en cuanto a comunicaciones, procesamiento, etc.

Es imprescindible que el desarrollador de la parte de networking conozca el coste de esas características de cara a diseñar un sistema de comunicaciones eficiente y que cumpla con las necesidades del videojuego. La decisión de qué tipo de comunicación quiere determina:

- El tipo de estructura del cliente y servidor.
- La eficiencia en las comunicaciones
- Otras características asociadas al tipo de servicio como pueden ser si es orientada a la conexión o no, comunicación mediante bytes o mensajes, capacidad de gestionar QoS o no, etc.

Esta decisión, a groso modo está relacionada con si queremos una comunicación confiable o no, es decir, en el caso de Internet, decidir entre usar un SOCK_STREAM o SOCK_DGRAM respectivamente y por ende, en la mayoría de los casos, entre TCP o UDP.

El uso de SOCK_STREAM implica una conexión confiable que lleva asociado unos mecanismos de reenvío, ordenación, etc. que consume tiempo de procesamiento a cambio de garantizar la entrega en el mismo orden de envío. En el caso de SOCK_DGRAM no se proporciona confiabilidad con lo que el tiempo de procesamiento se reduce.

¿Qué tipo de comunicación empleo?, bien, en función del tipo de datos que estemos transmitiendo deberemos usar un mecanismo u otro. Como norma general todos aquellos datos en tiempo real que no tiene sentido reenviar irán sobre SOCK_DGRAM mientras que aquellos tipos de datos mas sensibles y que son imprescindibles para el correcto funcionamiento deben ir sobre SOCK_STREAM.

Algunos ejemplos de uso de ambos protocolos nos pueden clarificar qué usos se le dan a uno y otro protocolo. En las últimamente populares conexiones multimedia en tiempo real, por ejemplo una comunicación VoIP, se suele usar para la transmisión del audio propiamente dicho el protocolo RTP (Real Time Protocol) que proporciona una conexión no confiable. Efectivamente, no tiene sentido reenviar una porción de audio que se pierde ya que no se puede reproducir fuera de orden y almacenar todo el audio hasta que el reenvío se produce generalmente no es aceptable en las comunicaciones. No obstante, en esas mismas comunicaciones el control de la sesión se delega en el protocolo (SIP (Session Initiation Protocol)) encargado del establecimiento de la llamada. Este tipo de datos requieren de confiabilidad y por ello SIP se implementa sobre TCP.

Podemos trasladar este tipo de decisiones al diseño de los módulos de networking de un determinado videojuego. Una vez decidida la información necesaria que se debe transmitir entre los diversos participantes, debemos clasificar y decidir, para cada flujo de información si va a ser mediante una conexión confiable no confiable.

Es difícil dar normas genéricas por que, como adelantamos en la introducción, cada videojuego es diferente, no obstante, generalmente se utilizará comunicaciones no confiables para:

- Aquella información que por su naturaleza, lleven el tiempo implícito en su validez como información. Generalmente toda la información de tiempo real como movimientos de granularidad fina , audio y vídeo, etc.
- Información que pueda ser inferida a partir de datos que recibimos antes o después. Por ejemplo, si el movimiento de un personaje se compone de varios movimientos intermedios y se puede inferir uno de esos movimiento intermedios a partir del resto (o incluso ignorar), esa información es candidata a ser transmitida mediante una conexión o confiable.

El resto de información que no cae en estas dos secciones puede ser susceptible de implementarse utilizando comunicaciones confiables.

Este primer paso de decisión de qué tipo de comunicación se necesita es importante ya que determina la estructura del cliente y el servidor.

29.10.2. Comunicaciones confiables

Las comunicaciones confiables garantizan la llegada de la información transmitida, sin pérdidas y la entrega al destino en el orden en el cual fueron enviados. La configuración por defecto de los *sockets* utiliza TCP y la estructura asociada a los clientes tiene la siguientes primitivas:

- *struct hostent *gethostbyname(const char *name)*: Esta estructura nos sirve para identificar al servidor donde nos vamos a conectar y puede ser un nombre (e.j. ejemplo.nombre.es), dirección IPv4 (e.j. 161.67.27.1) o IPv6. En el caso de error devuelve null.

- *int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen)*: Esta primitiva nos permite realizar la conexión con un servidor especificado por serv_addr usando el *socket* sockfd. Devuelve 0 si se ha tenido éxito.

- *ssize_t send(int s, const void *msg, size_t len, int flags)*: esta primitiva es utilizada para mandar mensajes a través de un *socket* indicando, el mensaje (*msg*), la longitud del mismo (*len*) y con unas opciones definidas en *flags*. Devuelve el número de caracteres enviados.

- *ssize_t recv(int s, void *buf, size_t lon, int flags)*: esta primitiva es utilizada para recibir mensajes a través de un *socket* ((s)), el mensaje es guardado en *buf*, se leen los bytes especificados por lon y con unas opciones definidas en *flags*. Devuelve el número de caracteres recibidos.

- *close(int sockfd)*: cierra un *socket* liberando los recursos asociados.

Con estas primitivas podemos escribir un cliente que se conecta a cualquier servidor e interacciona con él. Para la implementación del servidor tenemos las mismas primitivas de creación, envío, recepción y liberación. No obstante, necesitamos algunas primitivas mas:

- *int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen)*: Una vez creado un *socket* que nos va a servir de servidor necesitamos asociarle una dirección local (my_addr y su longitud addrlen).

- int listen(int s, int backlog): Con esta primitiva se indica que el *socket* s va a aceptar conexiones entrantes hasta un límite de conexiones pendientes definido en *backlog*.

- *int accept(int s, struct sockaddr *addr, socklen_t *addrlen)*: Esta función acepta una conexión pendiente, le asocia un nuevo *socket* que será devuelto por la función y que se empleará para la transmisión y recepción de datos propiamente dicho con el cliente que se acaba de aceptar. Llamaremos a este nuevo *socket*, *socket* de servicio.

Veamos una implementación de ejemplo que nos sirva para inscribir a un personaje en una partida gestionada por un servidor. En este primer ejemplo un usuario se desea registrar en un servidor de cara a enrolarse en una partida multijugador. La información de usuario será expresada por una estructura en C que podemos ver en el listado 29.1 conteniendo el nick del jugador, su tipo de cuenta y su clave. Este tipo de información es necesario transmitirlo por una conexión confiable ya que queremos que el proceso se realice de forma confiable.

Listado 29.1: Estructura de registro de jugadores

```
1 struct player{
2   char nick[10];
3   int account_type;
4   char passwd[4];
5 };
```

En las estructuras definidas para su transmisión en red no se deben usar punteros. Es necesario resaltar que los punteros sólo tienen sentido en el espacio de memoria local y al transmitirlo por la red, apuntan a datos distintos en la máquina destino. Salvo mecanismo que tenga en cuenta este aspecto, el uso de punteros no tiene sentido en este tipo de estructuras

La implementación relativa al proceso de comunicación por parte del servidor se muestra, a groso modo, en el listado 29.2. Por motivos didácticos y de simplificación no se ha introducido código de gestión de errores. Cada una de las invocaciones relativas a *sockets*, retornan códigos de error que el desarrollador debe tratar de forma adecuada.

En el servidor podemos ver la definición de las variables necesarias para las comunicaciones, incluidas las direcciones del propio servidor y del cliente que se le ha conectado. Definimos, con la constante PORT, el puerto donde estaremos a la escucha. Inicializaremos la dirección a la cual vamos a vincular el *socket* servidor de forma apropiada (lineas 17 a 20). Desde un punto de vista práctico en *sockets* de red, las direcciones locales y remotas se representan mediante la estructura *sockaddr_in*, donde existen tres campos cruciales, *sin_family* que indica el tipo de familia al cual pertenece la dirección, el *sin_port* que es el puerto y la estructura *sin_addr* dentro de la cual el campo *s_addr* nos indica la dirección IP remota a la cual vamos asociar al *socket*. Si indicamos, como en el ejemplo INADDR_ANY indicamos que aceptamos conexiones de cualquier dirección.

Para evitar problemas con el orden de los bytes (utilizando *bit_endian* o *little_indian*) se pasan a un formato de red utilizando las funciones *htons* y *htonl* que convierten del formato host, a un formato de red. Existen de igual manera *ntohl* y *ntohs* para realizar el proceso inverso. La l y la s al final de las funciones indican si es el tipo *sort* o *long*.

Listado 29.2: Servidor de registro de jugadores

```
1  int main(int argc, char *argv[])
2  {
3
4      struct sockaddr_in serverAddr;
5      int serverSocket;
6      struct sockaddr_in clientAddr;
7      int serviceSocket;
8      int read_bytes=0;
9      unsigned int serverPort;
10     unsigned int len_client=0;
11     struct player *new_player;
12
13
14     serverPort = PORT;
15     serverSocket = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
16
17     memset(&serverAddr, 0, sizeof(serverAddr));
18     serverAddr.sin_family = AF_INET;
19     serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
20     serverAddr.sin_port = htons(serverPort);
```

```
21
22      bind(serverSocket, (struct sockaddr *)&serverAddr, sizeof(
            serverAddr));
23      listen(serverSocket, MAXCONNECTS);
24      while(1)
25      {
26        printf("Listening clients..\n");
27        len_client = sizeof(clientAddr);
28        serviceSocket = accept(serverSocket, (struct sockaddr *) &
              clientAddr, &(len_client));
29        printf("Cliente %s conectado\n", inet_ntoa(clientAddr.
              sin_addr));
30        new_player = (struct player *)malloc(sizeof(struct player));
31        read_bytes= recv(serviceSocket, new_player,sizeof(struct
              player) ,0);
32        printf("Nuevo jugador registrado %s con cuenta %d y password
               %s\n", new_player->nick, new_player->account_type,
              new_player->passwd);
33        close(serviceSocket);
34      }
35
36  }
```

Una vez creado y vinculado un *socket* a una dirección, estamos listos para entrar en un bucle de servicio donde, se aceptan conexiones, se produce el intercambio de mensajes con el cliente conectado y con posterioridad se cierra el *socket* de servicio y se vuelven a aceptar conexiones. Si mientras se atiende a un cliente se intenta conectar otro, queda a la espera hasta un máximo especificado con la función *listen*. Si se supera ese máximo se rechazan las conexiones. En este caso la interacción se limita a leer los datos de subscripción del cliente.

El cliente, tal y como vemos en el listado 29.3, sigue unos pasos similares en cuanto a la creación del *socket* e inicialización de las direcciones. En este caso se debe introducir la dirección IP y puerto del servidor al cual nos queremos conectar. En este caso *localhost* indica que van a estar en la misma máquina (linea 19). A continuación nos conectamos y enviamos la información de subscripción.

En este ejemplo, el diseño del protocolo implementado no podría ser mas simple, la sintaxis de los paquetes están definidos por la estructura, la semántica es relativamente simple ya que el único paquete que se envía indican los datos de registro mientras que la temporización también es muy sencilla al ser un único paquete.

Listado 29.3: Cliente que registra un usuario en el servidor

```
1  int main(int argc, char *argv[])
2  {
3
4      struct sockaddr_in serverAddr;
5      struct hostent     *server_name;
6      int serviceSocket;
7      unsigned int serverPort;
8      struct player myData;
9      int send_bytes=0;
10     int len_server=0;
11
12     strncpy(myData.nick,"Carpanta",sizeof("Carpanta"));
13     myData.account_type = PREMIUM;
14      strncpy(myData.passwd,"123",sizeof("123"));
15
16     serverPort = PORT;
17     serviceSocket = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
18
19     server_name = gethostbyname("localhost");
20     memset(&serverAddr, 0, sizeof(serverAddr));
21     memcpy(&serverAddr.sin_addr, server_name->h_addr_list[0],
           server_name->h_length);
```

```
22      serverAddr.sin_family = AF_INET;
23      serverAddr.sin_port = htons(serverPort);
24
25      len_server = sizeof(serverAddr);
26      printf("Connecting to server..");
27      connect(serviceSocket,(struct sockaddr *) &serverAddr,(
            socklen_t)len_server);
28      printf("Done!\n Sending credentials.. %s\n", myData.nick);
29      send_bytes = send(serviceSocket, &myData, sizeof(myData),0);
30      printf("Done! %d\n",send_bytes);
31      close(serviceSocket);
32      return 0;
33  }
```

29.10.3. Comunicaciones no confiables

Información de estado

La información de estado que un servidor le envía a un cliente es sólo aquella información relativa a su visión parcial del estado del juego. Es decir, la mínima imprescindible para que el cliente pueda tener la información de qué le puede afectar en cada momento

En el caso de las comunicaciones no confiables, y tal y como vimos al principio de esta sección, se utilizan para enviar información cuya actualización debe ser muy eficiente y en la cual, la pérdida de un paquete o parte de la información, o no tiene sentido un proceso de reenvío o bien se puede inferir. Para mostrar este ejemplo podemos ver, para un juego FPS, como un cliente va a enviar información relativa a su posición y orientación, y el servidor le envía la posición y orientación de su enemigo.

Asumiremos que la información de la posición del enemigo siempre es relevante de cara a la representación del juego. En el listado 29.4 podemos ver la estructura *position* que nos sirve para situar un jugador en un juego tipo FPS. Este es un ejemplo de estructura que podemos asociar a comunicaciones no confiables ya que:

- Cada cambio en la posición y orientación del jugador debe generar una serie de mensajes al servidor que, a su vez, debe comunicarlo a todos los jugadores involucrados.
- Al ser un tipo de información que se modifica muy a menudo y vital para el juego, se debe notificar con mucha eficiencia.
- En principio la pérdida ocasional de algún mensaje no debería plantear mayores problemas si se diseña con una granularidad adecuada. La información perdida puede inferirse de eventos anteriores y posteriores o incluso ignorarse.

Listado 29.4: Estructura que representa la posición y orientación de un jugador

```
1 struct position{
2   int userID;
3   float x; // x coordenate
4   float y; // y coordenate
5   float z; // z coordenate
6   float rotation[3][3]; // 3x3 rotation matrix
7 };
```

El servidor (listado 29.5) en este caso prepara una estructura que representa la posición de un jugador (lineas 10-14), que podría ser un jugador manejado por la IA del juego, a continuación se crea el *socket* de servicio y se prepara para aceptar paquetes de cualquier cliente en el puerto 3000. Finalmente se vincula el *socket* a la dirección mediante la operación *bind* y se pone a la espera de recibir mensajes mediante la operación *recvfrom*. Una vez recibido un mensaje, en la dirección *clientAddr* tenemos la dirección del cliente, que podemos utilizar para enviarle mensajes (en este caso mediante la operación *sendto*).

Listado 29.5: Servidor de estado del juego

```
1  int main(int argc, char *argv[])
2  {
3    int serviceSocket;
4    unsigned int length;
5    struct hostent *server_name;
6    struct sockaddr_in localAddr, clientAddr;
7    struct position otherplayer;
8    struct position player;
9
10   otherplayer.userID=9;
11   otherplayer.x=10;
12   otherplayer.y=10;
13   otherplayer.z=0;
14   otherplayer.rotation[0][0]=2;
15
16   serviceSocket = socket( AF_INET, SOCK_DGRAM, 0 );
17   localAddr.sin_family = AF_INET;
18   localAddr.sin_port = htons(3000);
19   localAddr.sin_addr.s_addr = htonl(INADDR_ANY);
20
21   length = sizeof(clientAddr);
22   bind( serviceSocket, (struct sockaddr *)&localAddr, sizeof(
         localAddr) );
23   recvfrom(serviceSocket, &player, sizeof(struct position), 0, (
         struct sockaddr *)&clientAddr, &length );
24   printf("Player ID: %d is in position %f, %f, %f with orientation %f"
         ,player.userID,player.x, player.y, player.z, player.rotation
         [0][0]);
25   sendto(serviceSocket, &otherplayer, sizeof(struct position), 0, (
         struct sockaddr *)&clientAddr, length );
26   return 0;
27 }
```

El cliente en este caso guarda muchas similitudes con el servidor (listado 29.6). En primer lugar se prepara la estructura a enviar, se crea el *socket*, y se prepara la dirección de destino, en este caso, la del servidor que se encuentra en el mismo host (lineas 21-25), una vez realizada la vinculación se envía la posición y se queda a la espera de paquetes que provienen del servidor mediante la función *recvfrom*.

Listado 29.6: Cliente de un jugador

```
1  int main(int argc, char *argv[])
2  {
3    int serviceSocket;
4    unsigned int length;
5    struct hostent *server_name;
6    struct sockaddr_in localAddr, serverAddr;
7    struct position mypos;
8    struct position enemy;
9
10   mypos.userID=10;
11   mypos.x=23;
12   mypos.y=21;
13   mypos.z=0;
14   mypos.rotation[0][0]=1;
15
16   serviceSocket = socket( AF_INET, SOCK_DGRAM, 0 );
17   localAddr.sin_family = AF_INET;
18   localAddr.sin_port = 0;
19   localAddr.sin_addr.s_addr = htonl(INADDR_ANY);
20
21   server_name = gethostbyname("localhost");
22   memset(&serverAddr, 0, sizeof(serverAddr));
23   memcpy(&serverAddr.sin_addr, server_name->h_addr_list[0],
         server_name->h_length);
24   serverAddr.sin_family = AF_INET;
```

```
25    serverAddr.sin_port = htons(3000);
26
27    memset( &( serverAddr.sin_zero), '\0', 8 );
28    length = sizeof( struct sockaddr_in );
29    bind( serviceSocket, (struct sockaddr *)&localAddr, length );
30    sendto(serviceSocket, &mypos, sizeof(struct position), 0, (struct
          sockaddr *)&serverAddr, length );
31    recvfrom(serviceSocket, &enemy, sizeof(struct position), 0, (
          struct sockaddr *)&serverAddr, &length );
32    printf("My enemy ID: %d is in position %f, %f, %f with orientation %
          f\n",enemy.userID,enemy.x, enemy.y, enemy.z, enemy.rotation
          [0][0]);
33    return 0;
34  }
```

A lo largo de esta sección hemos visto las funciones básicas de envío y recepción mediante *sockets* TCP/IP. Aunque ya tenemos la base para construir cualquier tipo de servidor y cliente, en la parte de servidores hace falta introducir algunas características adicionales de cara al desarrollo de servidores eficientes y con la capacidad de atender a varios clientes de forma simultánea.

29.11. *Sockets* TCP/IP avanzados

Cuando hablamos de programación multijugador es necesario la gestión de varios clientes de forma simultánea. Existen dos alternativas a la atención de peticiones simultáneas de diversos clientes:

- crear un nuevo proceso por cada petición. El *socket* servidor recibe una petición de conexión y crea un *socket* de servicio. Este *socket* de servicio se pasa a un proceso creado de forma expresa para atender la petición. Este mecanismo desde el punto de vista del diseño es muy intuitivo con la consiguiente ventaja en mantenibilidad. De igual manera, para sistemas que sólo tienen un sólo procesador, este sistema adolece de problemas de escalabilidad ya que el número de cambios de contexto, para números considerables de clientes, hace perder eficiencia a la solución.
- Utilizar un mecanismo que nos permita almacenar las peticiones y notificar cuando un cliente nos manda una petición al servidor.

En el caso de acumular peticiones y atenderlas de forma secuencial necesitamos utilizar un mecanismo que nos permita saber qué *sockets* necesitan ser atendidos. Para ello tenemos la función select. La función select tiene la siguiente definición:

*int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);*

Donde el primer argumento n, es el mayor descriptor de los monitorizados mas uno, timeout es un límite de tiempo de espera antes de que la función retorne, con 0 la función retorna de manera inmediata mientras que si se especifica NULL la función select es bloqueante. readfds, writefds y exceptfds son tres conjuntos de descriptores de archivos (*sockets*) donde se esperará para ver si hay datos de lectura (readfds), para escribir (writefds) o donde existen excepciones (exceptfds).

La función select nos permite realizar entrada/salida asíncrona

La idea del select es que cuando retorna en cada uno de esos conjuntos se encuentran los identificadores de los *sockets* de los cuales puedes leer, escribir o hay alguna excepción. Con las operaciones FD_ZERO, FD_SET, FD_CLR y FD_ISSET se limpian, añaden y eliminan descriptores y se pregunta por un descriptor concreto en cualquiera de los conjuntos respectivamente.

Supongamos que queremos atender distintos tipos de conexiones en varios *sockets*, en principio no hay un orden de llegada de los paquetes y por lo tanto, no podemos bloquearnos en uno sólo de los *sockets*. La función select nos puede ayudar. Podemos evolucionar nuestro servidor UDP desarrollado anteriormente para incluir la función select (listado 29.7).

Listado 29.7: Utilización del select en monitorización de *sockets*

```
1
2    fd_set readSet;
3    FD_ZERO(&readSet);
4    FD_SET(serviceSocket,&readSet);
5
6    while(select(serviceSocket+1, &readSet,0,0,NULL)){
7      if (FD_ISSET(serviceSocket,&readSet)){
8        recvfrom(serviceSocket, &player, sizeof(struct position), 0,
             (struct sockaddr *)&clientAddr, &length );
9        printf("Player ID: %d is in position %f, %f, %f with orientation
              %f",player.userID,player.x, player.y, player.z, player.
             rotation[0][0]);
10       sendto(serviceSocket, &otherplayer, sizeof(struct position),
             0, (struct sockaddr *)&clientAddr, length );
11     }
12   }
13   return 0;
14 }
```

Debemos tener en cuenta los siguientes puntos a la hora de utilizar la función select

- Debemos añadir a los conjuntos monitorizados por el select todos aquellos descriptores de archivos que creemos y deseemos usar, bien sea para lectura o para escritura.
- En el caso de TCP, si añadimos el *socket* al cual se nos conectan los clientes, debemos añadir el *socket* de servicio creado al aceptar esa conexión a los conjuntos monitorizados si queremos hacer un seguimiento de este *socket*.
- El select nos retorna un número de descriptores en los cuales hay datos nuevos.
- Después del select debemos comprobar con FD_ISSET qué descriptor concreto se ha originado.
- Los descriptores no tienen por que ser solamente de *sockets*, pueden ser también descriptores de archivos.

Un aspecto a tener en cuenta con la función select es la semántica asociada a la estructura *timeval*. Algunos sistemas operativas actualizan el valor almacenado en esa estructura para reflejar la cantidad de tiempo que quedaba antes del timeout asociado. El problema es que otros sistemas operativos no actualizan este valor por lo que esta semántica puede variar.

29.11.1. Operaciones bloqueantes

De las funciones que estamos viendo, muchas de ellas son bloqueantes, es decir, se invocan y duermen el hilo que la invoca hasta que algo relevante sucede. Por ejemplo, el accept es una función bloqueante, recv, recvfrom, etc.

Mas que las funciones, la propiedad de que las operaciones sean o no bloqueantes está asociada a la creación del *socket*. Por defecto en la creación de un *socket*, se establece la propiedad de que las operaciones relacionadas con ese descriptor de *socket* sean bloqueantes. Para cambiar esta propiedad, debemos hacer uso de la función *int fcntl(int fd, int cmd, long arg);* que coge como argumento un descriptor de archivo (fd), un comando y una serie de opciones. La función fcntl se dedica a los descriptores de archivo en general, para establecer un *socket* no bloqueante hay que invocarla con el identificador del *socket*, FS_SETFL para el comando y O_NONBLOCK como argumento.

Las operaciones realizadas sobre el *socket* no serán bloqueantes por lo que habrá que gestionar apropiadamente esta condición y consultar el estado de los *sockets* de forma continua. Esta configuración es muy útil, por ejemplo, para hacer periódico el acceso de las comunicaciones haciéndolas mas predecibles y mientras seguir realizando otras tareas.

Hay que tener en cuenta que, en entornos Linux, las operaciones bloqueantes duermen el hilo que las invoca mientras no pase algo asociado a la funcionalidad de la función bloqueante (por ejemplo, accept, recv, etc.). Si optamos por operaciones no bloqueantes y por ello nos vemos obligados a realizar continuas consultas, deberemos gestionar nosotros mismos nuestros periodos dormidos si no queremos consumir recursos de CPU de forma innecesaria.

De igual manera, el valor devuelto por las operaciones realizadas sobre *socket* no bloqueantes generalmente será un error si no hay nada interesante asociado (por ejemplo, no hay datos que leer o conexión que aceptar). El valor al cual se asocia *errno* es EWOULDBLOCK (no obstante, en función de la implementación y sistema operativo utilizado debe consultarse este valor).

29.11.2. Gestión de buffers

Los *buffers* son estructuras que se utilizan para almacenar información de forma temporal para su procesado. Son estructuras necesarias para muchos cometidos como puede ser asociar procesos que tienen distintas velocidades en el procesamiento de la información, almacenamiento temporal, etc.

Cuando realizamos operaciones con *sockets*, los datos pasan realmente por una sucesión de *buffers* en el kernel que permiten el tratamiento de flujos continuos de bytes mediante operaciones parciales. Por lo tanto cuando recibimos información de un *socket* puede que no se nos proporcione toda la información recibida al igual que cuando enviamos una información puede que no se mande de forma completa en la misma operación. Esta gestión de cuando procesar la información tiene que ver con decisiones del kernel.

En el caso de *send*, tenemos la operación *sendall* que, con la misma semántica que el *send*, nos permite enviar todo un *buffer*.

Existen otros casos en el cual la gestión de *buffers* puede ralentizar las prestaciones ya que se necesitan copiar de forma continua. Por ejemplo si leemos de un archivo, la operación *read* sobre un descriptor de archivo almacena la información en un *buffer* interno al kernel que es copiado al *buffer* en espacio de usuario. Si a continuación queremos enviar esa información por un *sockets*, invocaríamos la función *send* y esa función copiaría ese *buffer*, de nuevo a un *buffer* interno para su procesado. Este continuo trasiego de información ralentiza la operación si lo que queremos es transmitir por completo el archivo. Lo ideal sería que las dos operaciones entre los *buffers* internos al *buffer* de aplicación se convirtiera en una única operación entre *buffers* internos.

Un ejemplo de función que implementa la filosofía zero-copy y lidia con el problema descrito de copia entre *buffers* es la operación *transferTo(..)* del paquete *java.nio.channels.FileChannel*. De igual forma algunos sistemas operativos *ix tienen una llamada al sistema sendfile(..) que tiene la misma semántica.

La técnica que minimiza las copias innecesarias entre *buffers* se denomina *zero-copy* y, a menudo, puede obtener una considerable mejora de las prestaciones. Es difícil estudiar esta técnica en profundidad porque depende del sistema operativo y de la aplicación concreta.

29.11.3. Serialización de datos

La amplia variedad de sistemas y configuraciones posibles hacen que la representación de datos y portabilidad de cualquier programa sea un aspecto importante a tener en cuenta en el desarrollo de sistemas multiplataforma. La programación con *sockets* no escapa a este problema y, al ser una programación de bajo nivel, tenemos que ocuparnos de aspectos como la representación de datos binarios en distintas plataformas y como transmitir este tipo de datos. A este nivel la principal distinción es el ordenamiento de los bytes (*little-endian* y *big-endian*).

En los listados anteriores, esta característica la omitíamos a excepción de la utilización de las funciones *htons()* y *ntohs()* que utilizamos para pasar a una codificación común denominada *Network Byte Order*. Estas funciones nos permiten lidiar con un formato común en muchas plataformas y las podemos utilizar para la gestión de enteros y flotantes:

```
1 #include <arpa/inet.h>
2 uint32_t htonl(uint32_t hostlong);
3 uint16_t htons(uint16_t hostshort);
4 uint32_t ntohl(uint32_t netlong);
5 uint16_t ntohs(uint16_t netshort);
```

Si queremos trabajar con ordenaciones concretas deberíamos consultar el archivo endian.h

A mas alto nivel se puede usar XDR (eXternal Data Representation), un estándar para la descripción y representación de datos utilizado por ONC (Open Network Computing)-RPC (Remote Procedure Call) pero el cual se puede usar de forma directa mediante las librerías adecuadas. Como veremos mas adelante, una de las principales labores que nos ahorra el uso de middlewares es precisamente, la gestión de la representación de datos.

29.11.4. Multicast y Broadcast

Los modelos de comunicación vistos hasta ahora están asociados a una comunicación cliente-servidor uno a uno. Este tipo de comunicaciones entre pares iguales se denomina unicast. Existen otros paradigmas de comunicación como son las comunicaciones uno a muchos (*Multicast*) y uno a todos (*Broadcast*). En el caso de comunicaciones *Multicast*, el servidor comunica a un grupo de clientes identificados por un identificador (en IP existen un grupo de direcciones especiales para identificar estos grupos) de forma común algún dato de su interés. Este tipo de comunicación es mucho mas efectiva ya que substituye, para un grupo de n clientes, a n conexiones *unicast*. En el caso de las comunicaciones *Broadcast* la comunicación se establece dentro de una subred entre el servidor y todos los computadores de ese dominio. Por motivos obvios las comunicaciones *broadcast* nunca atraviesan un *router*.

Para implementar la comunicación multicast se debe subscribir el cliente y el servidor a una dirección de grupo común. El rango de direcciones IPv4 para grupos multicast va desde 224.0.0.0 hasta la 239.255.255.255. Es necesario consultar la página web de la IANA (Internet Assigned Numbers Authority) ya que muchas de esas direcciones están asignadas a grupos preestablecidos.

El envío de un mensaje a un grupo multicast no se diferencia, en el código fuente, salvo por la dirección a la cual se envía que pertenece al rango anteriormente descrito. En la parte del receptor hay ligeras variaciones con dos objetivos principales:

- Abrir un *socket* que pertenece a un grupo multicast.
- Indicarle al sistema operativo que se una al grupo para recibir sus mensajes.

Para abrir un *socket* que pertenece a un grupo multicast:

```
1 setsockopt(fd,SOL_SOCKET,SO_REUSEADDR,&argument,sizeof(argument))
```

en esta llamada a *setsockopt* (que nos ayuda a establecer las opciones de los *socket*) donde básicamente se especifica que para una dirección multicast, vamos a reutilizar la dirección compartiendo varios *socket* el mismo puerto. En este caso *argument* es un u_int con un valor de 1. Faltaría la segunda parte:

```
1   struct ip_mreq multreq;
2   multreq.imr_multiaddr.s_addr=inet_addr("224.0.1.115");
3   multreq.imr_interface.s_addr=htonl(INADDR_ANY);
4   if (setsockopt(fd,IPPROTO_IP,IP_ADD_MEMBERSHIP,&multreq,sizeof(
        multreq)) < 0) {
5     perror("setsockopt");
6     exit(1);
7   }
```

donde de nuevo se indica mediante *setsockopt*, que el kernel se añada al grupo especificado en la estructura multreq. Con la función *getsockopt* podríamos consultar las opciones del *socket*.

La emisión en broadcast sería de forma similar:

```
1 setsockopt(sock, SOL_SOCKET, SO_BROADCAST, (void *) &argument,
2            sizeof(argument))
```

De nuevo, *argument* es un u_int con un valor 1 e indicamos que vamos a utilizar una dirección broadcast. Las direcciones broadcast tienen la parte de host puesto a uno y dependen de la red/subred.

No obstante, desde el punto de vista de la programación de videojuegos, los mensajes *broadcast* no tienen tanta utilidad como los mensajes multicast ya que, como dijimos anteriormente, los paquetes broadcast se quedan en el dominio de difusión por lo que no llegarían a jugadores fuera de la red/subred donde son emitidos.

29.11.5. Opciones de los *sockets*

Aunque a lo largo de estas secciones hemos visto algunas de las opciones, representamos a continuación un breve resumen de las opciones mas comunes que podrían ser de interés en la implementación con *sockets*:

- SO_DONTROUTE: Evitar el uso de la tabla de rutas
- SO_ERROR: Identifica el error relacionado con el *socket*.
- SO_KEEPALIVE :Comprobación periódica de conexión.
- SO_LINGER: Indica qué hacer con los datos pendientes una vez cerrada la conexión.
- SO_RCVLOWAT: Tamaño en bytes que un *socket* debe recibir para que select reaccione. El valor por defecto es 1.
- SO_REUSEADDR: Reutilización de la dirección de máquina.
- SO_SNDLOWAT: lo mismo que SO_RCVLOWAT pero en envío.
- SO_RCVBUF: Indica el tamaño en bytes del *buffer* de recepción
- SO_SNDBUF: Especifica el tamaño en bytes del buffer de envío en conexiones confiables.
- SO_SNDTIMEO: timeout de envío.
- SO_RCVTIMEO: timeout de recepción.
- SO_BROADCAST: Permitir mensajes broadcast.

Aunque la implementación con *sockets* es muy eficiente, el programador debe realizar un diseño e implementación de bajo nivel. Existen alternativas que nos permiten elevar el nivel de abstracción de las comunicaciones. Estas alternativas serán objeto de estudio en las siguientes secciones.

29.12. Middlewares de comunicaciones

Un middleware de comunicaciones es un sofisticado sistema de IPC (Inter Process Communication) orientado a mensajes. A diferencia de los otros IPC como los sockets, los middlewares suelen ofrecer soporte para interfaces concretas entre las entidades que se comunican, es decir, permiten definir estructura y semántica para los mensajes.

Existen muchísimos middlewares: RPC (Remote Procedure Call), CORBA, Java RMI (Remote Method Invocation), .Net Remoting, etc. En todos ellos, el programador puede especificar un API. En el caso de RPC se indican un conjunto de funciones que podrán ser invocadas remotamente por un cliente. Los demás, y la mayoría de los actuales, son RMI (Remote Method Invocation), es decir, son middlewares orientados a objetos. La figura 29.6 muestra el esquema de invocación remota a través de un núcleo de comunicaciones típico de este tipo de middlewares.

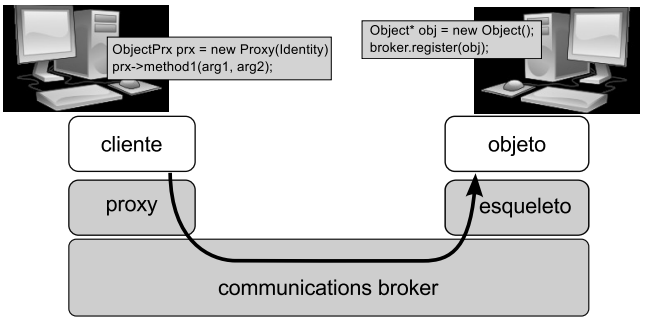


Figura 29.6: Invocación a método remoto

Objeto distribuido

Es un objeto cuyos métodos (algunos al menos) pueden ser invocados remotamente.

En un diseño orientado a objetos, el ingeniero puede decidir qué entidades del dominio serán accesibles remotamente. Puede «particionar» su diseño, eligiendo qué objetos irán en cada nodo, cómo se comunicarán con los demás, cuáles serán los flujos de información y sus tipos. En definitiva está diseñando una aplicación distribuida. Obviamente todo eso tiene un coste, debe tener muy claro que una invocación remota puede suponer hasta 100 veces más tiempo que una invocación local convencional.

Un videojuego en red, incluso sencillo, se puede diseñar como una aplicación distribuida. En este capítulo veremos cómo comunicar por red las distintas partes del juego de un modo mucho más flexible, cómodo y potente que con sockets.

Obviamente, el middleware se basa en las mismas primitivas del sistema operativo y el subsistema de red. El middleware no puede hacer nada que no se pueda hacer con sockets. Pero hay una gran diferencia; con el middleware lo haremos con mucho menos esfuerzo gracias a las abstracciones y servicios que proporciona, hasta el punto que habría muchísimas funcionalidades que serían prohibitivas en tiempo y esfuerzo sin el middleware. El middleware encapsula técnicas de programación de sockets y gestión de concurrencia que pueden ser realmente complejas de aprender, implementar y depurar; y que con él podemos aprovechar fácilmente. El middleware se encarga de identificar y numerar los mensajes, comprobar duplicados, retransmisiones, comprobaciones de conectividad, asignación de puertos, gestión del ciclo de vida de las conexiones, despachado asíncrono y un largo etcétera.

29.12.1. ZeroC Ice

ICE (Internet Communication Engine) es un middleware de comunicaciones orientado a objetos desarrollado por la empresa ZeroC Inc[2]. Ya hemos visto algunas utilidades de ICE en los capítulos de concurrencia y patrones. Aquí lo utilizaremos con su finalidad principal, y la que le da el nombre.

ICE soporta múltiples lenguajes (Java, C#, C++, ObjectiveC, Python, Ruby, PHP, etc.) y multiplataforma (Windows, GNU/Linux, Solaris, Mac OS X, Android, iOS, etc.) lo que proporciona una gran flexibilidad para construir sistemas muy heterogéneos o integrar sistemas existentes.

Además ofrece *servicios comunes* muy valiosos para la propagación de eventos, persistencia, tolerancia a fallos, seguridad,

29.12.2. Especificación de interfaces

Cuando nos planteamos una interacción con un objeto remoto, lo primero es definir el «contrato», es decir, el protocolo concreto que cliente y objeto (servidor) van a utilizar para comunicarse.

Antes de las RPC cada nueva aplicación implicaba definir un nuevo protocolo (independientemente si es público o no) y programar las rutinas de serialización y des-serialización de los parámetros de los mensajes para convertirlos en secuencias de bytes, que es lo que realmente podemos enviar a través de los sockets. Esta tarea puede ser compleja porque se tiene que concretar la ordenación de bytes, el padding para datos estructurados, las diferencias entre arquitecturas, etc.

Los middleware permiten especificar la interfaz mediante un lenguaje de programación de estilo declarativo. A partir de dicha especificación, un compilador genera código que encapsula toda la lógica necesaria para (des)serializar los mensajes específicos produciendo una representación externa canónica de los datos. A menudo el compilador también genera «esqueletos» para el código dependiente del problema. El ingeniero únicamente tiene que rellenar la implementación de los funciones o métodos.

El lenguaje de especificación de interfaces de Ice se llama SLICE (Specification Language for Ice) y proporciona compiladores de interfaces (*translators*) para todos los lenguajes soportados dado que el código generado tendrá que compilar/enlazar con el código que aporte el programador de la aplicación. En cierto sentido, el compilador de interfaces es un generador de protocolos a medida para nuestra aplicación.

SLICE

El lenguaje SLICE (Specification Language for Ice) al igual que otros muchos lenguajes para definición de interfaces (como IDL (Interface Definition Language) o XDR) son puramente declarativos, es decir, no permiten especificar lógica ni funcionalidad, únicamente interfaces.

29.12.3. Terminología

Conviene definir algunos conceptos básicos que se utilizan habitualmente en el desarrollo de sistemas distribuidos:

Servidor
: Es la entidad pasiva, normalmente un programa que inicializa y activa los recursos necesarios para poner objetos a disposición de los clientes.

Objeto
: Los objetos son entidades abstractas que se identifican con un identidad, que debería ser única al menos en la aplicación. Los objetos implementan al menos una interfaz remota definida en una especificación Slice.

[2] `http://www.zeroc.com`

Sirviente
Es el código concreto que respalda la funcionalidad del objeto, es quién realmente hace el trabajo.

Cliente
Es la entidad activa, la que solicita servicios de un objeto remoto mediante invocaciones a sus métodos.

Proxy
Es un representante de un objeto remoto. El cliente en realidad interacciona (invoca) a un *proxy*, y éste se encarga con la ayuda del núcleo de comunicaciones, de llevar el mensaje hasta el objeto remoto y después devolverle la respuesta al cliente.

Nótese que *servidor* y *cliente* son roles en la comunicación, no tipos de programas. Es bastante frecuente que un mismo programa sirva objetos a la vez que invoca servicios de otros.

La figura 29.7 muestra los componentes principales del middleware y su relación en una aplicación típica que involucra a un cliente y a un servidor. Los componentes de color azul son proporcionados en forma de librerías o servicios. Los componentes marcados en naranja son generados por el compilador de interfaces.

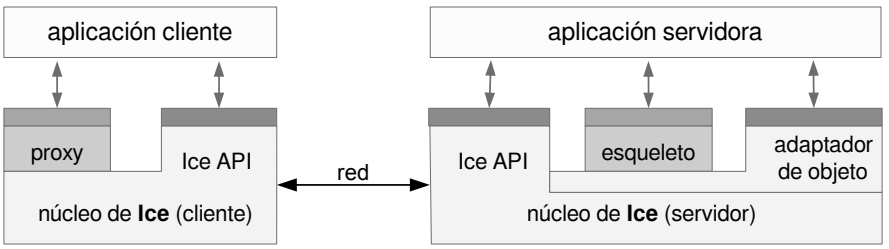


Figura 29.7: Componentes básicos del middleware

El diagrama de secuencia de la figura 29.8 describe una interacción completa correspondiente a una invocación remota síncrona, es decir, el cliente queda bloqueado hasta que la respuesta llega de vuelta. En el diagrama, el cliente efectúa una invocación local convencional sobre un método sobre el proxy. El proxy funciona como una referencia remota al objeto distribuido alojado en el servidor y por ello implementa la misma interfaz. El proxy serializa la invocación y construye un mensaje que será enviado al host servidor mediante un socket. Al tratarse de una llamada síncrona, el proxy queda a la espera de la respuesta lo que bloquea por tanto a la aplicación cliente.

Ya en el nodo servidor, el esqueleto recibe el mensaje, lo des-serializa, identifica el objeto destino y sintetiza una llamada equivalente a la que realizó al cliente. A continuación realiza una invocación local convencional a un método del objeto destino y recoge el valor de retorno. Lo serializa en un mensaje de respuesta y lo envía de vuelta al nodo cliente. El proxy recoge ese mensaje y devuelve el valor de retorno al cliente, completando la ilusión de una invocación convencional.

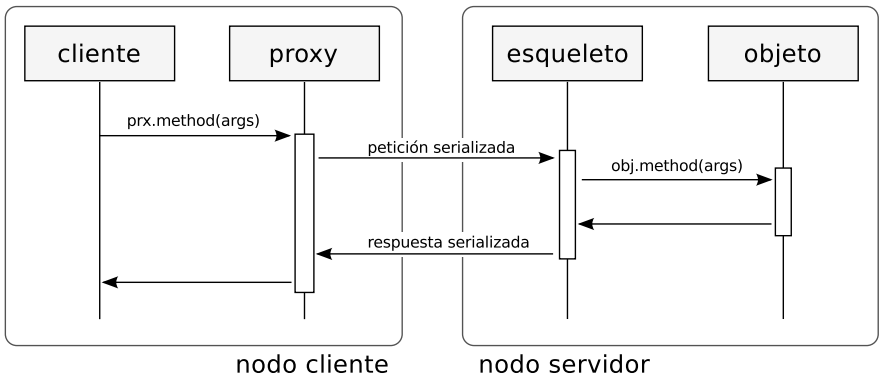


Figura 29.8: Diagrama de secuencia de una invocación a un objeto remoto

29.12.4. «Hola mundo» distribuido

En esta aplicación, el servidor proporciona un objeto que dispone del método remoto `puts()` que imprimirá en la salida estándar la cadena que el cliente pase como parámetro.

Tal como hemos visto, lo primero que necesitamos es escribir la especificación de la interfaz remota de nuestros objetos. El siguiente listado corresponde al fichero `Hello.ice` y contiene la interfaz *Hello* en lenguaje SLICE.

Listado 29.8: Especificación SLICE para un «Hola mundo» distribuido: `Hello.ice`

```
1 module Example {
2   interface Hello {
3     void puts(string message);
4   };
5 };
```

Lo más importante de este fichero es la declaración del método `puts()`. El compilador de interfaces generará un esqueleto que incluirá una versión del la interfaz *Hello* en el lenguaje de programación que el programador decida. Cualquier clase que herede de es interfaz *Hello* deberá definir un método `puts()` con la misma signatura, que podrá ser invocado remotamente. De hecho, en la misma aplicación distribuida puede haber varias implementaciones del mismo interfaz incluso en diferentes lenguajes.

El compilador también genera el proxy para el cliente y, del mismo modo, clientes escritos en distintos lenguajes o sobre distintas arquitecturas podrán usar los objetos remotos que cumplan la misma interfaz.

Para generar proxy y esqueleto en C++ (para este ejemplo) usamos el *translator* `slice2cpp`.

```
$ slice2cpp Hello.ice
```

Esto genera dos ficheros llamados `Hello.cpp` y `Hello.h` que deben ser compilador para obtener las aplicaciones cliente y servidor.

Sirviente

El compilador de interfaces ha generado una clase *Example::Hello*. La implementación del *sirviente* debe heredar de esa interfaz proporcionando una implementación (por sobrecarga) de los métodos especificados en la interfaz SLICE.

El propio compilador de interfaces puede generar una clase «hueca» que sirva al programador como punto de partida para implementar el sirviente:

```
$ slice2cpp --impl Hello.ice
```

De este modo genera además los ficheros `HelloI.cpp` y `HelloI.h`. La letra `'I'` hace referencia a «**I**mplementación del interfaz». El fichero de cabecera generado (`HelloI.h`) tiene el siguiente aspecto:

Listado 29.9: Sirviente de la aplicación Hello: `Hello.h`

```
1  #ifndef __HelloI_h__
2  #define __HelloI_h__
3
4  #include <Hello.h>
5
6  namespace Example {
7
8    class HelloI : virtual public Hello {
9    public:
10     virtual void puts(const ::std::string&,
11                       const Ice::Current&);
12   };
13 }
14
15 #endif
```

Y el fichero de implementación `HelloI.cpp`:

Listado 29.10: Sirviente de la aplicación Hello: `Hello.cpp`

```
1  #include <iostream>
2  #include "HelloI.h"
3
4  void
5  Example::HelloI::puts(const ::std::string& message,
6                        const Ice::Current& current) {
7    std::cout << message << std::endl;
8  }
```

La única modificación respecto al fichero generado es la línea 9.

Servidor

Nuestro servidor consiste principalmente en la implementación de una clase que herede de *Ice::Application*. De ese modo ahorramos parte del trabajo de inicialización del `communicator`[3]. En esta clase debemos definir el método `run()`. Vea el listado 29.11.

[3] El `communicator` representa el *broker* de objetos del núcleo de comunicaciones.

Listado 29.11: Servidor de la aplicación Hello: `Server.cpp`

```
1  #include <Ice/Ice.h>
2  #include "HelloI.h"
3
4  using namespace std;
5  using namespace Ice;
6
7  class Server: public Application {
8    int run(int argc, char* argv[]) {
9      Example::HelloPtr servant = new Example::HelloI();
10
11     ObjectAdapterPtr adapter = \
12         communicator()->createObjectAdapter("HelloAdapter");
13     ObjectPrx proxy = adapter->add(
14         servant, communicator()->stringToIdentity("hello1"));
15
16     cout << communicator()->proxyToString(proxy) << endl;
17
18     adapter->activate();
19     shutdownOnInterrupt();
20     communicator()->waitForShutdown();
21
22     return 0;
23   }
24 };
25
26 int main(int argc, char* argv[]) {
27   Server app;
28   return app.main(argc, argv);
29 }
```

En la línea 9 se crea el sirviente (una instancia de la clase *HelloI*). Las líneas 11–12 crea un adaptador de objetos, que es el componente encargado de multiplexar entre los objetos alojados en el servidor. El adaptador requiere un endpoint —un punto de conexión a la red materializado por un protocolo (TCP o UDP), un host y un puerto. En este caso esa información se extrae de un fichero de configuración a partir de el nombre del adaptador (`HelloAdapter` en este caso).

En las lineas 13–14 registramos el sirviente en el adaptador mediante el método `add()` indicando para ello la identidad que tendrá el objeto (`hello1`). En este ejemplo la identidad es un identificador bastante simple, pero lo recomendable es utilizar una secuencia globalmente única. El método `add()` devuelve una referencia al objeto distribuido recién creado, que llamamos `proxy`. La línea 17 imprime en consola su representación textual. A partir de esa representación el cliente podrá obtener a su vez un proxy para el objeto remoto.

La línea 18 es la activación del adaptador, que se ejecuta en otro hilo. A partir de ese momento el servidor puede escuchar y procesar peticiones para sus objetos. El método `waitForShutown()` invocado en la línea 20 bloquea el hilo principal hasta que el comunicador sea terminado. El método `shutdownOnInterrupt()`, que se invoca en la línea 19, le indica a la aplicación que termine el comunicador al recibir la señal SIGQUIT (Control-C). Por último, las líneas 26–29 contienen la función `main()` en la que se crea una instancia de la clase `Server` y se invoca su método `main()`.

Cliente

La aplicación cliente únicamente debe conseguir una referencia al objeto remoto e invocar el método `puts()`. El cliente también se puede implementar como una especialización de la clase *Ice::Application*. El código completo del cliente aparece a continuación:

Listado 29.12: Cliente de la aplicación Hello: `Cliente.cpp`

```
1  #include <Ice/Ice.h>
2  #include "Hello.h"
3
4  using namespace Ice;
5  using namespace Example;
6
7  class Client: public Ice::Application {
8    int run(int argc, char* argv[]) {
9      ObjectPrx proxy = communicator()->stringToProxy(argv[1]);
10     HelloPrx hello = HelloPrx::checkedCast(proxy);
11
12     hello->puts("Hello, World!");
13
14     return 0;
15   }
16 };
17
18 int main(int argc, char* argv[]) {
19   Client app;
20   return app.main(argc, argv);
21 }
```

El programa acepta por línea de comandos la representación textual del proxy del objeto remoto. A partir de ella se obtiene un objeto proxy (línea 8). Sin embargo esa referencia es para un proxy genérico. Para poder invocar los métodos de la interfaz *Hello* se requiere una referencia de tipo `HelloPrx`, es decir, un proxy a un objeto remoto *Hello*.

Para lograrlo debemos realizar un moldeado del puntero (*downcasting*[4]) mediante el método estático `HelloPrx::checkedCast()` (línea 9). Gracias al soporte de introspección de los objetos remotos, Ice puede comprobar si efectivamente el objeto remoto es del tipo al que tratamos de convertirlo. Esta comprobación no se realizaría si empleáramos la modalidad `uncheckedCast()`.

Una vez conseguido el proxy del tipo correcto (objeto `hello`) podemos invocar el método remoto `puts()` (línea 12) pasando por parámetro la cadena `"Hello, World!"`.

Compilación

La figura 29.9 muestra el esquema de compilación del cliente y servidor a partir del fichero de especificación de la interfaz. El ficheros marcados en amarillo corresponden a aquellos que el programador debe escribir o completar, mientras que los ficheros generados aparecen en naranja.

Para automatizar el proceso de compilación utilizamos un fichero `Makefile`, que se muestra en el listado 29.13

[4]Consiste en moldear un puntero o referencia de una clase a una de sus subclases asumiendo que realmente es una instancia de ese tipo.

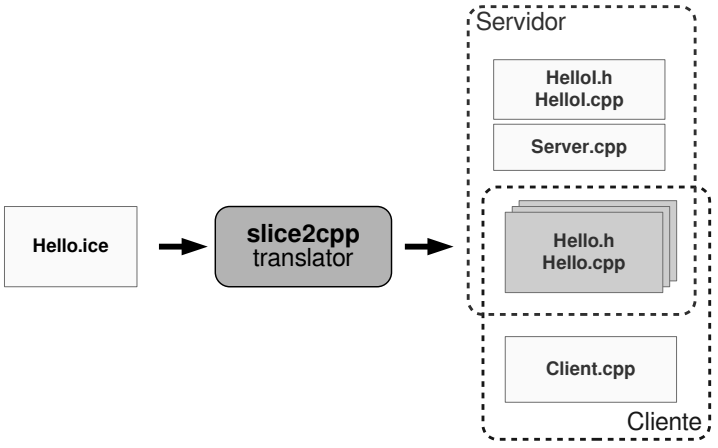


Figura 29.9: Esquema de compilación de cliente y servidor

Listado 29.13: Aplicación Hello: `Makefile`

```
1  CC=g++
2  CXXFLAGS=-I.
3  LDLIBS=-lIce -lIceUtil
4
5  APP=Hello
6  STUBS=$(addprefix $(APP), .h .cpp)
7
8  all: Server Client
9
10 Server: Server.o $(APP)I.o $(APP).o
11 Client: Client.o $(APP).o
12
13 Server.cpp Client.cpp: $(STUBS)
14
15 %.cpp %.h: %.ice
16     slice2cpp $<
17
18 clean:
19     $(RM) Server Client $(STUBS) *.o *~
20     $(RM) *.bz2 IcePatch2.sum
```

Ejecutando el servidor

Si ejecutamos el servidor obtendremos un error:

```
$ ./Server
!! 03/10/12 19:52:05.733 ./Server: error: ObjectAdapterI.cpp:915: Ice
    ::InitializationException:
   initialization exception:
   object adapter `HelloAdapter' requires configuration
```

Como vimos en la sección 29.12.4, el servidor necesita información específica que le indique los *endpoint* en los que debe escuchar el adaptador `HelloAdapter`. Para ello se debe proporcionar escribir un fichero adicional (`hello.cfg`) cuyo contenido aparece en el siguiente listado:

Listado 29.14: Servidor de la aplicación Hello: `hello.cfg`

```
1 HelloAdapter.Endpoints=tcp -p 9090
```

Este tipo de ficheros contiene definiciones de *propiedades*, que son parejas clave=valor. La mayoría de los servicios de Ice puede configurarse por medio de propiedades, lo que le otorga gran flexibilidad sin necesidad de recompilar el código. Esta propiedad en concreto indica que el adaptador debe utilizar un socket TCP en el puerto 9090.

Para que el servidor cargue las propiedades del fichero de configuración ejecuta:

```
$ ./Server --Ice.Config=hello.cfg
hello1 -t:tcp -h 192.168.0.12 -p 9090:tcp -h 10.1.1.10 -p 9090
```

En esta ocasión el programa arranca sin problemas y queda ocupando la shell como corresponde a cualquier servidor. Lo que aparece en consola es, como ya vimos, la representación textual del proxy para el objeto distribuido. La línea contiene varios datos:

- La identidad del objeto: `hello1`.
- El tipo de proxy (`-t`), que corresponde a *twoway*. Existen otros tipos que implican semánticas de llamada diferentes: `-o` para *oneway*, `-d` para *datagram*, etc.
- Una lista de endpoints separados con el carácter `:`, que corresponden con sockets. Para cada uno de ellos aparece el protocolo, la dirección IP indicada con el parámetro `-h` y el puerto indicado con el parámetro `-p`.

Concretamente, el adaptador de nuestro servidor escucha en el puerto TCP de dos interfaces de red puesto que el fichero de configuración no lo limitaba a una interfaz concreta.

Ejecutando el cliente

Para ejecutar el cliente debemos indicar el proxy del objeto remoto en línea de comandos, precisamente el dato que imprime el servidor. Nótese que se debe escribir entre comillas para que sea interpretado como un único parámetro del programa:

```
$ ./Client "hello1 -t:tcp -h 192.168.0.12 -p 9090"
```

El programa se ejecuta y retorna inmediatamente y veremos como la cadena *Hello, World!* aparece en la consola del servidor.

29.12.5. *twoway*, *oneway* y *datagram*

ICE puede realizar diferentes tipos de invocaciones:

- twoway: Son las que ICE trata de realizar por defecto. La invocación twoway implica el uso de un protocolo de transporte confiable (TCP) y es obligatorio si el método invocado tiene un valor de retorno.
- oneway: Las invocaciones *oneway* sólo pueden utilizarse cuando los métodos no retornan ningún valor (`void`) y no tienen parámetros de salida. También requieren de un protocolo confiable, pero en determinadas circunstancias pueden fallar la entrega.
- datagram: Igual que las oneway únicamente pueden utilizase para invocar métodos sin valor de retorno y sin parámetros de salida. Utilizan un protocolo de transporte no confiable (UDP). A su vez eso implica que el tamaño del mensaje está limitado al máximo de un paquete IP (64KiB) y sufran de todas la otras limitaciones de UDP: no hay control de orden, duplicados ni garantía de entrega.

La principal diferencia entre las invocaciones twoway con respecto a los otros tipos es que el cliente recupera el control del flujo de ejecución tan pronto como el mensaje de invocación es enviado, sin necesidad de esperar ningún tipo de confirmación o respuesta desde el servidor (ver figura 29.10).

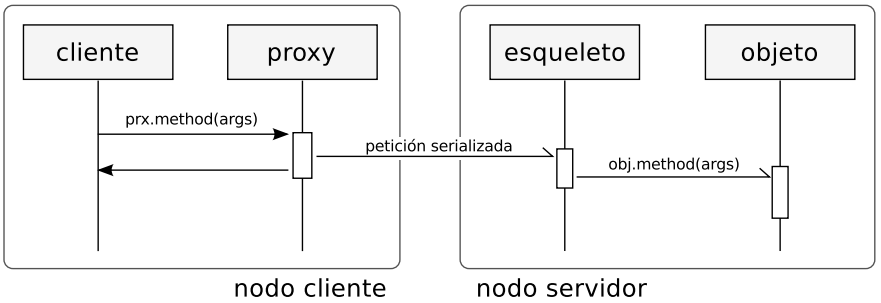


Figura 29.10: Diagrama de secuencia de una invocación datagram

A partir de un proxy válido es posible obtener proxies adicionales que pueden utilizarse para efectuar los distintos tipos de llamadas, siempre que el adaptador disponga de los endpoints adecuados. Para ello, el proxy dispone de los métodos `ice_twoway()`, `ice_oneway()` e `ice_datagram()`.

Existen otras dos modalidades de proxy: *batched oneway*, que se obtiene con el método `ice_batchOneway()`, y *batched datagram* que se maneja a través del método `ice_batchDatagram()`. Estos proxies pueden almacenar temporalmente los mensajes de invocación dirigidos al mismo destino, agruparlos en un solo segmento y enviarlos como una sola entidad. Este procedimiento aumenta sensiblemente la eficiencia y el servidor los procesa del mismo modo que si se tratara de invocaciones convencionales.

29.12.6. Invocación asíncrona

Una llamada síncrona o bloqueante (lo habitual en un middleware RMI) implica una espera explícita no determinista, algo que obviamente degrada gravemente el desempeño de una aplicación que debe tener un algo grado de inmediatez. La solución tradicional cuando este tipo de interacción con el servidor es ineludible es realizar estas invocaciones en un hilo de ejecución diferente, de modo que el hilo principal, que se encarga de actualizar la interfaz gráfica y de leer la entrada del usuario, puede seguir progresando normalmente.

Este mecanismo lo incorpora ICE de forma transparente, de modo que el programador no se tiene que preocupar del manejo de los hilos ni el tratamiento de las respuestas. El programador puede realizar llamadas remotas a los mismos métodos definidos en la interfaz utilizando un procedimiento diferente que involucra un objeto que permite obtener el estado de la invocación y, eventualmente, obtener el resultado de la operación. A este mecanismo es a lo que denominamos invocación asíncrona.

De hecho ICE proporciona varias modalidades para realizar invocaciones asíncronas. La que vemos aquí es probablemente la más limpia y se basa en la creación de objetos *callback*. Para realizar la invocación remota, el cliente utiliza un método del proxy con el prefijo `begin` que acepta un parámetro adicional: el *callback object*. Esta invocación retorna inmediatamente y el cliente puede seguir ejecutando otras operaciones.

Cuando la invocación remota se ha completado, el núcleo de ejecución de la parte del cliente invoca el método indicado en el objeto *callback* pasado inicialmente, proporcionando los resultados de la invocación. En caso de error, invoca otro método en el que proporciona información sobre la excepción asociada.

En el listado 29.15 se muestra el código completo para el cliente que crea y proporciona un *callback* según la técnica indicada.

Listado 29.15: Cliente AMI que recibe la respuesta mediante un *callback* (`cpp-ami/Client-callback.cpp`)

```
#include <Ice/Ice.h>
#include <factorial.h>

using namespace std;
using namespace Ice;
using namespace Example;

class FactorialCB : public IceUtil::Shared {
public:
  void response(const Ice::Long retval) {
    cout << "Callback: Value is " << retval << endl;
  }

  void failure(const Exception& ex) {
    cout << "Exception is: " << ex << endl;
  }
};

class Client: public Ice::Application {
public:
  int run(int argc, char* argv[]) {

    ObjectPrx proxy = communicator()->stringToProxy(argv[1]);
    MathPrx math = MathPrx::checkedCast(proxy);

    Callback_Math_factorialPtr factorial_cb =
      newCallback_Math_factorial(new FactorialCB,
                 &FactorialCB::response,
                 &FactorialCB::failure);

    math->begin_factorial(atoi(argv[2]), factorial_cb);
    return 0;
  }

};

int main(int argc, char* argv[]) {
  if (argc != 3) {
    cerr << "usage: " << argv[0] << "<server> <value>" << endl;
    return 1;
  }
```

```
42
43    Client app;
44    return app.main(argc, argv);
45 }
```

Las [8-17] corresponden con la definición de la clase *callback* `FactorialCB`. El método `response()` será invocado por el núcleo de ejecución cuando el método remoto haya terminado y el resultado esté de vuelta. También se define un método `failure()` que será invocado con una excepción en el caso de producirse.

En la [26] se crea la instancia del *callback*, mediante una función específica para este método: `newCallback_Math_factorial()` pasándole una instancia de nuestra clase `FactorialCB` y sendos punteros a los métodos definidos en ella. Esa función y todos los tipos nombrados siguiendo el patrón `Callback_{interface}_{método}` son generados automáticamente por el compilador de interfaces en el espacio de nombre *Example*.

Por último en la [31] aparece la invocación al método remoto pasando la instancia del *callback* (`factorial_cb`) como segundo parámetro.

29.12.7. Propagación de eventos

Uno de los servicios más sencillos y útiles de ICE es ICESTORM. Se trata de un servicio de propagación de eventos, pero a diferencia de otros servicios similares en otros middlewares, ICESTORM propaga invocaciones en lugar de datos. Es a todos los efectos una implementación distribuida del patrón de diseño publicación/subscripción u observador.

Para utilizar el servicio de eventos se requiere un proxy a un *TopicManager*. Este objeto permite listar, crear y destruir canales (*topics*). Los canales también son objetos, que residen en el mismo servidor en el que se encuentre el *TopicManager*.

Para recibir eventos procedentes de un canal necesitamos subscribir un objeto. Para enviar invocaciones a un canal necesitamos el proxy al *publicador del canal*. El publicador es un objeto especial que no tiene interfaz concreta. Podemos invocar cualquier método sobre él. El canal enviará esa invocación a todos sus subscriptores. Sin embargo, si el subscriptor no soporta la invocación elevará una excepción y el canal lo des-suscribirá inmediatamente. Por esa razón debemos crear canales diferentes para cada interfaz de la que queramos propagar eventos.

Los canales solo pueden propagar invocaciones oneway o datagram, es decir, que no tengan valores de retorno, dado que sería complejo tratar las respuestas de todos los subscriptores. El canal es un intermediario que desacopla completamente a publicadores y suscriptores. Todos conocen el canal, pero no se conocen entre sí.

Veamos su funcionamiento sobre un ejemplo. Crearemos un canal de eventos para la interfaz *Exam::Hello* del ejemplo anterior. Empezamos por codificar el subscriptor; es básicamente un servidor: crea un sirviente y un adaptador, y lo añade el primero al segundo (líneas 39–42) del siguiente listado. El código completo para el subscriptor aparece el en siguiente listado:

Listado 29.16: Subscriptor

```
1 #include <Ice/Application.h>
2 #include <IceStorm/IceStorm.h>
3 #include <IceUtil/UUID.h>
4 #include "Hello.h"
5
6 using namespace std;
7 using namespace Ice;
```

```
 8 using namespace IceStorm;
 9 using namespace Example;
10
11
12 class HelloI : public Hello {
13   void puts(const string& s, const Current& current) {
14     cout << "Event received: " << s << endl;
15   }
16 };
17
18 class Subscriber : public Application {
19   TopicManagerPrx get_topic_manager() {
20     string key = "IceStormAdmin.TopicManager.Default";
21     ObjectPrx base = communicator()->propertyToProxy(key);
22     if (!base) {
23       cerr << "property " << key << " not set." << endl;
24       return 0;
25     }
26
27     cout << "Using IceStorm in: '" << key << "' " << endl;
28     return TopicManagerPrx::checkedCast(base);
29   }
30
31 public:
32   virtual int run(int argc, char* argv[]) {
33     TopicManagerPrx topic_mgr = get_topic_manager();
34     if (!topic_mgr) {
35       cerr << appName() << ": invalid proxy" << endl;
36       return EXIT_FAILURE;
37     }
38
39     ObjectPtr servant = new HelloI;
40     ObjectAdapterPtr adapter = \
41       communicator()->createObjectAdapter("Hello.Subscriber");
42     ObjectPrx base = adapter->addWithUUID(servant);
43
44     TopicPrx topic;
45     try {
46       topic = topic_mgr->retrieve("HelloTopic");
47       topic->subscribeAndGetPublisher(QoS(), base);
48     }
49     catch(const NoSuchTopic& e) {
50       cerr << appName() << ": " << e
51        << " name: " << e.name << endl;
52       return EXIT_FAILURE;
53     }
54
55     cout << "Waiting events... " << base << endl;
56
57     adapter->activate();
58     shutdownOnInterrupt();
59     communicator()->waitForShutdown();
60
61     topic->unsubscribe(base);
62
63     return EXIT_SUCCESS;
64   }
65 };
66
67 int main(int argc, char* argv[]) {
68   Subscriber app;
69   return app.main(argc, argv);
70 }
```

Podemos destacar la obtención del proxy al *TopicManager* (línea 33), que se delega a un método privado (`get_topic_manager()`).

Después se obtiene una referencia al canal `"HelloTopic"` (línea 46), que debería existir. Si no existe se produce una excepción y el programa termina. Si el canal existe, se subscribe el objeto (linea 47). Por último se activa el adaptador y el servidor queda a la espera de recibir eventos como en ejemplo anterior.

Si el subscriptor es un objeto, el publicador es un cliente. En este caso le hemos dado al publicador la responsabilidad de crear el canal si no existe (líneas 10–13 del siguiente listado). Solo se muestra el método `run()` puesto que el resto del código es prácticamente idéntico.

Lo más interesante es la obtención del publicador del canal a partir de la referencia al canal (línea 19) y el *downcast* para poder invocar sobre él método de la interfaz *Example::Hello*. Nótese que este molde usa la modalidad `uncheckedCast()` dado que el publicador no implementa realmente ninguna interfaz.

Por último utilizamos el proxy `hello` para enviar diez eventos con la cadena `"Hello World!"`.

Listado 29.17: Publicador

```
1   virtual int run(int argc, char*[]) {
2     TopicManagerPrx topic_mgr = get_topic_manager();
3     if (!topic_mgr) {
4       cerr << appName() << ": invalid proxy" << endl;
5       return EXIT_FAILURE;
6     }
7
8     TopicPrx topic;
9     try {
10      topic = topic_mgr->retrieve("HelloTopic");
11    } catch (const NoSuchTopic& e) {
12      cerr << appName()
13       << ": no sucho topic found, creating" << endl;
14      topic = topic_mgr->create("HelloTopic");
15    }
16
17    assert(topic);
18
19    ObjectPrx prx = topic->getPublisher();
20    HelloPrx hello = HelloPrx::uncheckedCast(prx);
21
22    cout << "publishing 10 'Hello World' events" << endl;
23    for (int i = 0; i < 10; ++i)
24      hello->puts("Hello World!");
25
26    return EXIT_SUCCESS;
```

Arranque del servicio

IceStorm está implementado como un servicio ICE. Los servicios son librería dinámicas que deben ser lanzadas con el servidor de aplicaciones cuyo nombre es IceBox. Vemos la configuración de IceBox en el siguiente listado:

Listado 29.18: Configuración de IceBox para lanzar IceStorm

```
1 IceBox.Service.IceStorm=IceStormService,34:createIceStorm --Ice.
      Config=icestorm.config
2 IceBox.ServiceManager.Endpoints=tcp -p 7000
```

Sólo se definen dos propiedades: La carga del servicio IceStorm y los endpoints del gestor remoto de IceBox. La configuración de IceStorm a la que se hace referencia (`icestorm.config`) es la siguiente:

Listado 29.19: Configuración de IceStorm

```
1 IceStorm.TopicManager.Endpoints=tcp -p 10000
2 IceStormAdmin.TopicManager.Default=IceStorm/TopicManager:tcp -p
      10000
3 IceStorm.Publish.Endpoints=tcp -p 2000
4 Freeze.DbEnv.IceStorm.DbHome=db
5
6 Hello.Subscriber.Endpoints=tcp
```

Las líneas 1–3 especifican los endpoints para los adaptadores del *TopicManager*, el objeto de administración y los publicadores de los canales. La línea 4 es el nombre del directorio donde se almacenar la configuración del servicio (que es persistente automáticamente). La línea 6 es el endpoint del adaptador del publicador, que se ha incluido en este fichero por simplicidad aunque no es parte de la configuración de IceStorm.

Una vez configurado podemos lanzar el servicio y probar el ejemplo. Lo primero es arrancar el icebox (en *background*):

```
$ icebox --Ice.Config=icebox.config &
```

A continuación podemos usar la herramienta de administración en línea de comando para crear el canal, asumiendo que no existe:

```
$ icestormadmin --Ice.Config=icestorm.config -e "create HelloTopic"
```

Es momento de arrancar el suscriptor:

```
$ ./subscriber --Ice.Config=icestorm.config
```

Y por último en una consola diferente ejecutamos el publicador:

```
$ ./publisher --Ice.Config=icestorm.config
```

Hecho lo cual veremos los eventos llegar al suscriptor:

```
$ ./subscriber --Ice.Config=icestorm.config
Event received: Hello World!
Event received: Hello World!
Event received: Hello World!
Event received: Hello World!
Event received: Hello World!
Event received: Hello World!
Event received: Hello World!
Event received: Hello World!
Event received: Hello World!
Event received: Hello World!
```

Este servicio puede resultar muy útil para gestionar los eventos de actualización en juegos distribuidos. Resulta sencillo crear canales (incluso por objeto o jugador) para informar a los interesados de sus posición o evolución. El acoplamiento mínimo que podemos conseguir entre los participantes resulta muy conveniente porque permite que arranquen en cualquier orden y sin importar el número de ellos, aunque obviamente una gran cantidad de participantes puede degradar el desempeño de la red.

También resulta muy interesante que los eventos estén modelados como invocaciones. De ese modo es posible eliminar temporalmente el canal de eventos e invocar directamente a un solo subscriptor durante las fases iniciales de modelado del protocolo y desarrollo de prototipos.

Capítulo 30

Sonido y Multimedia

Guillermo Simmross Wattenberg
Francisco Jurado Monroy

En este capítulo se discuten dos aspectos que son esenciales a la hora de afrontar el desarrollo de un videojuego desde el punto de vista de la inmersión del jugador. Uno de ellos está relacionado con la edición de audio, mientras que el otro está asociado a la integración y gestión de escenas de vídeo dentro del propio juego.

El **apartado sonoro** es un elemento fundamental para dotar de realismo a un juego, y en él se aglutinan elementos esenciales como son los efectos sonoros y elementos que acompañan a la evolución del juego, como por ejemplo la banda sonora. En este contexto, la primera parte de esta capítulo describe la importancia del audio en el proceso de desarrollo de videojuegos, haciendo especial hincapié en las herramientas utilizadas, el proceso creativo, las técnicas de creación y aspectos básicos de programación de audio.

Respecto a la **integración de vídeo**, en la segunda parte del capítulo se muestra la problemática existente y la importancia de integrar escenas de vídeo dentro del propio juego. Dicha integración contribuye a la evolución del juego y fomenta la inmersión del jugador en el mismo.

30.1. Edición de Audio

30.1.1. Introducción

Videojuegos, música y sonido

Tanto la música como los efectos sonoros son parte fundamental de un videojuego, pero en ocasiones el oyente no es consciente de la capacidad que ambos tienen para cambiar por completo la experiencia lúdica. Una buena combinación de estos elementos aumenta enormemente la percepción que el usuario tiene de estar inmerso en la acción que sucede en el juego. La **capacidad de la música** para cambiar el estado de ánimo del oyente es clave en el papel que ejerce en un videojuego: permite transportar al jugador al mundo en el que se desarrolla la acción, influirle en su percepción del contenido visual y transformar sus emociones.

Proceso de inmersión

La música es una herramienta única para hacer que el jugador se implique, viva y disfrute del videojuego.

Dependiendo del momento del juego, de la situación y de las imágenes que muestre nuestro juego, la música debe ser acorde con las sensaciones que se desee generar en el oyente: en un juego de estrategia militar, se optará por utilizar instrumentos musicales que nos evoquen al ejército: cornetas, tambores… si el juego es de estrategia militar futurista, sería buena idea combinar los instrumentos mencionados con otros instrumentos sintéticos que nos sugieran estar en un mundo del futuro.

Pero la música no es el único recurso disponible que permite llevar al jugador a otras realidades, los efectos de sonido también son una pieza fundamental en la construcción de realidades virtuales.

La realidad cotidiana de nuestras vidas está repleta de sonidos. Todos ellos son consecuencia de un cambio de presión en el aire producido por la vibración de una fuente. En un videojuego se intenta que el jugador reciba un estimulo similar al de la realidad cuando realiza una acción. Cualquier sonido real se puede transportar a un videojuego, pero ¿y los sonidos que no existen? ¿cómo suena un dragón? y lo más interesante ¿cómo se crean los sonidos que emitiría un dragón? ¿serían diferentes esos sonidos dependiendo del aspecto del dragón? El trabajo del creador de audio en un videojuego es hacer que el jugador crea lo que ve, y un buen diseño del sonido en un videojuego ayudará de forma definitiva a conseguir este objetivo.

El artista, por tanto, debe utilizar una buena combinación de las dos herramientas de las que dispone —música y efectos sonoros— para conseguir ambientar, divertir, emocionar y hacer vivir al jugador la historia y la acción que propone el videojuego.

Interactividad y no linealidad

Elaborar el sonido de un videojuego tiene puntos en común a elaborar el de una película, pero dos características fundamentales de los videojuegos hacen que el planteamiento de una labor y otra sean totalmente diferentes: la interactividad y la no linealidad.

La interactividad en un videojuego es la capacidad que este tiene de reaccionar a acciones realizadas por el jugador.

Las acciones que se llevan a cabo al jugar a un videojuego pueden ser pulsaciones de teclas, movimientos del ratón o del joystick, movimientos de un volante o de mando inalámbrico y hoy en día hasta movimientos corporales del propio jugador. Todas estas acciones pueden producir una reacción en un videojuego.

La no linealidad en un videojuego es la capacidad que este tiene de reaccionar a largo plazo a acciones realizadas por el jugador.

La no linealidad en un videojuego es consecuencia de su interactividad: subir los impuestos en un simulador de ciudades puede aumentar el descontento de la población a largo plazo.

En el cine, una **banda sonora** es siempre de principio a fin una obra estática e invariante, independientemente del momento en el que el músico la componga (antes o después de rodar la película): en el montaje final es donde se decidirá en qué momento sonará cada pieza.

En un videojuego el artista debe componer pensando en los diferentes ambientes que puede haber en el videojuego, en si las situaciones serán relajadas o de acción trepidante, y muchas veces con una exigencia extra sobre la composición para el cine: la acción está por determinar. En la mayoría de los casos no se podrá saber de antemano cuánto tiempo tiene que durar la pieza musical, lo cual obliga al autor a hacer que pueda repetirse indefinidamente sin cansar al jugador, o a componer diferentes piezas que puedan encajar entre sí sin que el jugador note un corte brusco.

Lo mismo que pasa con la música sucederá con los efectos sonoros de forma más pronunciada. En el cine, los **efectos sonoros** siempre se montan sobre lo ya grabado por la cámara. Esto permite conocer de antemano desde qué ángulo ve el espectador al elemento que produce el sonido en cada momento. Diseñar el sonido que produciría un avión al despegar en una escena de una película se reduce a un solo sonido que responda exactamente a la imagen (ya sea un sonido estereofónico o envolvente).

Por el contrario, diseñar el sonido que se debe producir en un videojuego conlleva trabajar con el programador del motor de sonido para conocer las exigencias técnicas de éste y conocer qué sonidos serán necesarios y qué técnicas habrá que aplicar. Muy probablemente será necesaria la utilización de diferentes técnicas de tratamiento de audio y la creación de varios sonidos que tengan continuidad entre sí y puedan repetirse indefinidamente sin que lo parezca.

En contraposición a lo dicho hasta ahora, en muchos juegos actuales existen escenas no interactivas, en las que la forma de trabajar será muy similar a la utilizada en la creación de audio para películas.

El creador de audio para videojuegos debe conocer el funcionamiento interno del videojuego en lo que se refiere a la forma que este tiene de tratar y generar el sonido.

Por todo lo anterior, la composición musical y el diseño de sonido para videojuegos requiere de ciertas aptitudes complementarias sobre la creación de audio general.

La música en la industria del videojuego

En el **proceso de creación** de música y efectos sonoros para videojuegos no solo influyen los factores relacionados directa y estrictamente con materias musicales y de sonido. Este proceso es solo una parte del proyecto que supone la creación de un videojuego, cuyo tamaño, presupuesto, objetivo y fecha de entrega pueden variar en función de otras muchas características.

Dependiendo del tamaño de la compañía y del proyecto, una misma persona puede ejecutar varios de los roles típicos de este equipo: diseñador de sonido, encargado de diálogos, director de contratación, compositores y programadores de audio.

El volumen de negocio del mercado español de videojuegos se incrementó un 5 % en 2010, hasta situarse en 1.000 millones de euros, gracias al impulso de los juegos online y los videojuegos para móviles. Las previsiones para el ejercicio 2011-2012 indican que se alcanzarán los 1.225 millones de euros al final del periodo.

El **compositor** se encarga de la creación de la música del videojuego y en muchas ocasiones de orquestar esta música. También suele encargarse de contratar y dirigir las grabaciones en directo que se necesiten para el juego. El diseñador de sonido trabaja conjuntamente con los desarrolladores de herramientas sonoras del videojuego, creando, integrando y supervisando las tomas de audio y las librerías de sonido que se utilizarán, que pueden ser libres, compradas a una empresa externa o elaboradas por la propia compañía para su futura utilización en otros proyectos.

El **encargado de los diálogos** se ocupa de contratar, grabar y dirigir a los actores que pondrán su voz a los personajes del juego. Los programadores de audio desarrollan las herramientas de audio e integran los sonidos en el juego, tomando muchas veces el rol de diseñadores de sonido. Las herramientas que programan pueden ser externas al videojuego -para facilitar la integración de las grabaciones de audio- o internas, formando parte del motor de sonido del videojuego, que a su vez se integrará con los demás motores: el de inteligencia artificial, el de gráficos y el de física.

Por lo general, no se comienza a trabajar en el sonido de un videojuego hasta que están bien sentadas las bases del diseño y se ha comenzado el desarrollo.

Durante el proceso de desarrollo de un videojuego, es muy probable que evolucione en cierta manera la visión del videojuego que tengan el jefe de proyecto o el director artístico y se vean obligados a replantear la ambientación y aspecto de este, y por tanto las exigencias para la banda sonora o los efectos de sonido. Es una situación muy común que piezas musicales que ya se habían aceptado como integrantes del videojuego tengan que ser recompuestas, adaptadas, recortadas, reorquestadas, reinstrumentalizadas o en el peor de los casos descartadas para satisfacer las nuevas características del juego o de un nuevo giro en su planteamiento.

Por todo lo anterior, el **músico y diseñador de sonido** debe cumplir una serie de aptitudes muy especiales para ser capaz de desarrollar tareas muy diversas, tanto técnicas como artísticas, para las que se necesitan conocimientos variados. Además de estas aptitudes, debe ser creativo, innovador, perseverante, trabajar rápido y ser meticuloso.

30.1.2. Herramientas para la creación de audio

Hoy día el ordenador es la parte central del conjunto de elementos necesarios para construir un estudio de creación musical. En él se realizan todas las tareas de secuenciación, edición del sonido, grabación multipista, masterización y exportación a diferentes formatos.

Dependiendo del software que se prefiera, determinaremos las capacidades mínimas necesarias de nuestro ordenador. Eso sí, si deseamos que las tareas se realicen con fluidez, cuanto más potente sea la máquina, mayor productividad tendremos. Un procesador más rápido, más memoria RAM y un disco duro de gran capacidad nos facilitarán la vida enormemente, ya que los ficheros de audio suelen ocupar mucho espacio y las tareas de procesamiento necesitan de altas velocidades de proceso si se desea que lleven poco tiempo (o incluso puedan aplicarse y modificarse en tiempo real).

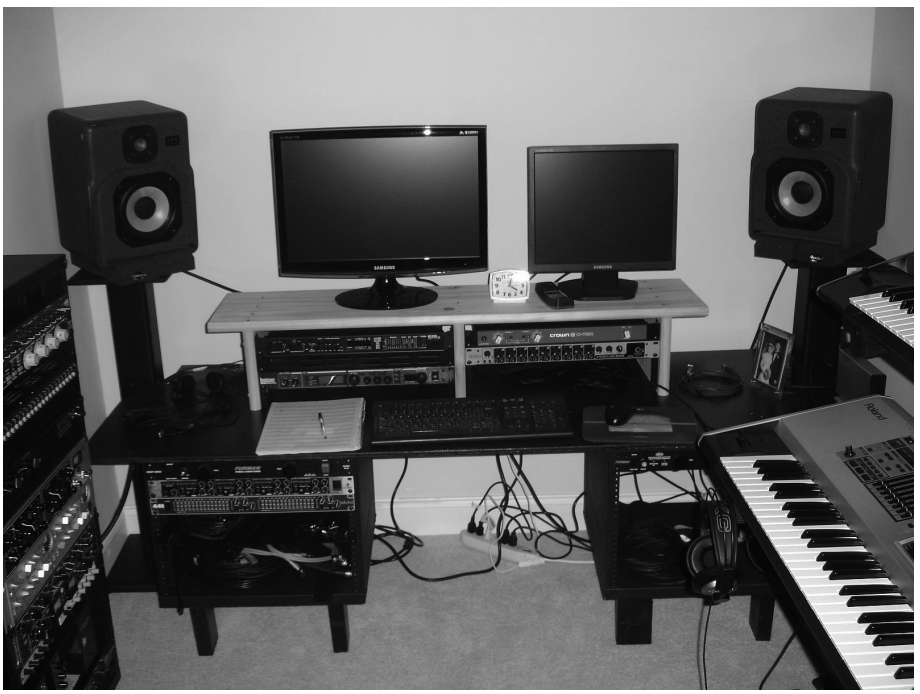


Figura 30.1: Para comenzar a crear un estudio casero sólo necesitaremos un ordenador con una buena tarjeta de sonido. Poco a poco podremos complementarlo con instrumentos reales, diferente hardware y nuevo software.

Secuenciad. multipistas

Muchos secuenciadores son también multipistas en los que se pueden crear pistas de audio además de pistas MIDI. Esto permite mezclar instrumentos reales con instrumentos generados en el ordenador.

Software

La herramienta principal con la que contamos hoy día para la creación de audio es el software musical. Existen multitud de programas de audio en el mercado con muy diferentes utilidades:

- **Secuenciadores**: son programas capaces de grabar y reproducir una serie de eventos MIDI, asignándoles diferentes muestras de sonido. Gracias a este tipo de software podremos memorizar los eventos MIDI procedentes de diferentes instrumentos. Ejemplos de este tipo de software son Linux Multimedia Studio (LMMS), Rosegarden, Steinberg Cubase, Cakewalk Sonar, Avid Pro Tools, Apple Logic o Propellerheads Reason.
- **Multipistas**: permite reproducir y controlar diferentes instrumentos o sonidos de forma independiente o conjunta, facilitando las labores de mezcla. Ejemplos de este tipo de software son Ardour, Digidesign Pro Tools, Steinberg Cubase, Cakewalk Sonar, Adobe Audition y Sony Vegas.
- **Editores**: permiten grabar, manipular y procesar archivos de sonido provenientes de ficheros en diferentes formatos y guardarlos en otros formatos. Las dos características más importantes de estos programas -la edición y la conversión a diferentes formatos- son esenciales en el mundo de la creación de audio para videojuegos. Ejemplos de este tipo de software son Audacity, Adobe Audition o Sony Sound Forge.
- **Software de masterización**: permiten retocar y ajustar un grupo de ficheros de sonido para que todos ellos estén igualados en volumen y funcionen como un único conjunto de piezas de audio. Aunque este tipo de software está más orientado a la producción de discos musicales, puede ser también útil para que, una vez creado el material sonoro que se incluirá en el videojuego, adaptemos sus niveles y retoquemos por última vez la ecualización y compresión en caso de ser necesario. Entre este tipo de programas podemos encontrar Sony CD Architect, Steinberg Wavelab o T-RackS.
- **Plug-ins**: son pequeñas extensiones de la funcionalidad de un programa anfitrión. Por lo general, en el ámbito de la música por ordenador, los plug-ins suelen ser módulos de efectos para el procesado de sonido o instrumentos virtuales. Hoy día existen todo tipo de plug-ins para añadir reverberación, delay, procesado dinámico, modulación, efectos de guitarra, filtrado, etc y para simular sintetizadores e instrumentos reales.
- **Software de loops**: este tipo de software permite crear música superponiendo diferentes bloques que pueden repetirse una y otra vez. Estos programas resultan útiles para crear bucles que serán incluidos en otras piezas musicales. Ejemplos de este tipo de software son Sony Acid, Propellerhead Recycle, Ableton Live o FL Studio.

SW libre y audio

Existen multitud de alternativas de software libre para la creación de audio: LMMS (secuenciación y software de loops), Ardour, energyXT, Qtractor (secuenciación, multipistas y masterización), Audacity (edición), mda plugin collection (plug-ins) y otros muchos.

Hardware

El apoyo necesario para facilitar la composición y la configuración del software de audio puede conseguirse con diferentes tipos de hardware:

- **Instrumentos y controladores MIDI**: el instrumento por excelencia para dinamizar la comunicación con el ordenador es el teclado. La casi totalidad de los teclados actuales son capaces de comunicarse con otros dispositivos mediante el protocolo MIDI. Gracias a él, la introducción de notas y la

composición en general será notablemente más fácil y fluida. Además de otros muchos tipos de instrumentos MIDI como baterías o guitarras, también pueden encontrarse controladores de diferentes formas y utilidades: pads, mesas de mezclas, trackballs, superficies...

- **Samplers**: son dispositivos electrónicos que pueden almacenar y modificar muestras de sonidos. El usuario puede cargar o grabar sonidos y crear tablas para asignarlos a notas musicales y poder así utilizar un teclado, un secuenciador u otro dispositivo para lanzar los sonidos almacenados.
- **Otros instrumentos**: además del teclado como instrumento central para componer música por ordenador, la utilización de instrumentos clásicos, especialmente los que son difíciles de emular digitalmente, como guitarras, vientos y voz, dará a nuestra música una nueva dimensión, aportando riqueza, humanidad y realismo a las piezas musicales que compongamos, además de abrirnos nuevas posibilidades y estimular nuestra creatividad.

Figura 30.2: Existen multitud de controladores MIDI en el mercado. En la imagen, Native Instruments Maschine (que es además un sampler y un secuenciador).

30.1.3. El proceso creativo

Actualmente las empresas desarrolladoras de videojuegos han realizado un esfuerzo considerable para adaptar la música a las diferentes situaciones de los videojuegos, creando tensión si algún suceso importante va a suceder en el desarrollo del juego o tranquilizando el estilo musical si la acción que se desarrolla se corresponde con un instante relajado de la historia.

Este enfoque no siempre es posible, y otros juegos simplemente se sirven de la música para crear una ambientación que se adapte al estilo del juego o para mantener su ritmo. El compositor de música para videojuegos debe ser capaz de dominar muchos de los procesos involucrados en la creación musical: compositor, arreglista, ingeniero, productor, músico, mezclador... Cada uno de estos procesos se llevan a cabo en diferentes fases del proceso creativo.

Preproducción

El proceso creativo comienza con la fase de preproducción, donde se comienza a elaborar el documento de diseño del juego. Este documento detalla todos los aspectos relevantes: historia, diálogos, mapas, sonido, gráficos, animaciones y programación. En este momento se fija el tipo de música y su papel en el videojuego y se elabora una lista que defina la música que tendrá cada parte del juego, si habrá un tema principal, si la música deberá cambiar dependiendo de ciertas variables del videojuego o cómo ciertos elementos musicales deben transmitir o corresponderse con detalles o situaciones concretas.

Producción

Una vez que el editor del juego aprueba el documento de diseño, el equipo de desarrollo comienza su trabajo. La persona o personas encargadas de crear el audio de un videojuego se incorpora al proyecto en las últimas fases de desarrollo (al finalizar la fase de producción normalmente). Hasta este momento, lo más probable es que el equipo de programadores habrá utilizado placeholders para los sonidos y su música favorita para acompañar al videojuego.

El creador de audio tiene que enfrentarse a la dura labor de crear sonidos y música lo suficientemente buenos como para conseguir que todo el equipo de desarrollo dé por buenos el nuevo material (aunque la decisión final no sea del equipo de desarrollo, pueden ejercer una gran influencia sobre el veredicto final).

En esta fase debe comprobarse tanto la validez técnica de los sonidos y música (si se adaptan a las especificaciones) como la calidad del material entregado. Es muy común que se cambien, repitan o se añadan sonidos y/o composiciones en este momento, por lo que el creador de audio debe estar preparado para estas contingencias (previéndolo en el presupuesto o creando a priori más material del entregado).

Al finalizar el proceso de producción, se determina cómo y en qué momentos se reproducirá qué archivo de sonido y con qué procesado en caso de utilizar procesamiento en tiempo real. Por lo general, y pese a lo que pudiera parecer en primera instancia, este proceso puede llevar hasta la mitad del tiempo dedicado a todo el conjunto de la producción.

Las consolas de última generación incluyen procesadores digitales de señal (DSPs) que permiten modificar en tiempo real efectos de sonido, o sus CPUs son lo suficientemente potentes como para hacer estos cálculos por sí mismas. A partir de un sonido de partida (un golpe metálico, por ejemplo), se puede generar una versión del mismo sonido con eco o reverberación (si el golpe se produjese en una caverna), ocluida (si el golpe se produjese detrás de una pared), enfatizada (si el golpe se produce más cerca del jugador), etc.

Actualmente ha aumentado la tendencia a la utilización de distintas APIs (paquetes de programación a modo de librerías de software), como son FMOD o ISACT, ya que disminuyen el tiempo necesario para llevar a cabo esta fase del proceso.

El posicionamiento del sonido en un **entorno tridimensional** también ha cobrado gran importancia en los últimos tiempos. La mayoría de las videoconsolas y ordenadores personales actuales soportan sonido surround, lo que permite crear un entorno más realista e inmersivo en los videojuegos. Este tipo de tecnologías dan una nueva

dimensión a los videojuegos y otra variable con la que jugar durante el proceso de diseño: el sonido tridimensional puede convertirse en otro elemento protagonista para el jugador, ya que puede permitirle posicionar cualquier fuente de sonido en un mundo virtual (un enemigo, un objeto, etc.).

Post-producción

Durante el proceso de post-producción se lleva a cabo la **masterización** y mezcla de los archivos de sonido una vez incluidos en el videojuego. Los sonidos y música que se reproducen durante una partida compiten simultáneamente por ser oídos por el jugador. Es necesario tener en cuenta qué sonidos son más o menos importantes en cada momento del juego, y asignar un volumen adecuado a cada uno de ellos.

Al realizar la mezcla de los diferentes sonidos creados, debe tenerse en cuenta el espectro de frecuencias de cada uno de los sonidos, para no acumular demasiados sonidos en un mismo rango frecuencial. El proceso de mezclado también puede ser dinámico y ser llevado a cabo automáticamente por el motor de sonido. Los efectos de sonidos pueden ser ecualizados o procesados en función de diferentes variables del videojuego, pero también puede modificarse dinámicamente una partitura añadiendo o eliminando instrumentos según lo que suceda en el juego.

Una forma práctica de preparar el mezclado final y evitar posibles problemas es decidir desde un principio —durante el proceso de pre-producción incluso— qué frecuencias se asignarán a cada tipo de efecto de sonido, a la música y a los diálogos. Durante la producción se tendrá en cuenta esto para ya asignar bandas de frecuencias durante el retoque de los sonidos, facilitando el posterior proceso de post-producción.

El **proceso de mezclado** no consiste en aumentar el volumen de determinados sonidos, sino de atenuar los menos importantes en cada momento. El problema es decidir cuál es la prioridad de la gran cantidad de sonidos que existen en un juego, incluida música y diálogos. Una vez más, la naturaleza de los videojuegos hace enormemente difícil la obtención de una solución óptima. La interactividad da un grado de libertad al jugador que permite que coincidan varios sonidos o partes de la música importantes con diálogos clave para el desarrollo del juego. Esto obliga al encargado de mezclar o de priorizar los sonidos a tomar la decisión menos mala o más óptima para los casos más probables.

El **proceso creativo de sonido** en los videojuegos no solo se refiere a la creación de audio, sino que una gran parte del tiempo se invierte en la implementación y acoplamiento de los sonidos y música creada. Esta implementación es especialmente importante, puesto que determina cómo y cuándo se reproducirán los sonidos durante el videojuego. Además, la complicación que supone generar una mezcla de sonidos adecuada hace aún más determinante esta parte del proceso creativo.

Por tanto, no solo debemos tener en cuenta la parte creativa y musical cuando pensemos en el audio de un videojuego, los aspectos técnicos también son muy importantes y harán que una buena pieza musical, unos efectos sonoros realistas y una buena grabación de diálogos destaquen adecuadamente en el videojuego.

30.1.4. Técnicas de creación

Interactividad e inmersión

El sonido y la música de un videojuego pueden tener diferentes funciones. En ciertos juegos, el sonido y tiene un papel fundamental, afectando directamente a la narrativa o al desarrollo del juego. En otros, la música es un acompañamiento a la acción y, aunque puede ser más o menos adaptativa, no constituye un elemento del juego en sí misma.

La música también puede utilizarse para realzar la estructura del videojuego, utilizando temas principales en momentos diferentes de la acción o ciertos leitmotifs. Los efectos sonoros pueden tener, además de su función normal de responder directamente a ciertas acciones del juego, una función simbólica. En algunos juegos, los sonidos son utilizados para dar pistas o dirigir al jugador hacia un lugar o para advertirle de un peligro inminente. También pueden utilizarse para centrar la atención del jugador en un elemento concreto de la acción.

Una **regla de oro** a tener en cuenta a la hora de crear efectos sonoros es que el sonido más real no es necesariamente más verosímil. Intuitivamente nuestro cerebro puede asociar sonidos a seres fantásticos (de los que no tiene por qué conocer su sonido) o asociar sonidos irreales a situaciones cotidianas (el golpe de un puñetazo en una película de acción). Lo anterior es aplicable también a la música: ¿qué música es más apropiada para un videojuego ambientado en un mundo virtual?

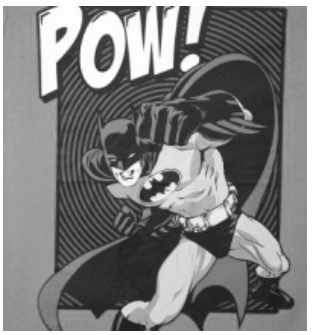


Figura 30.3: Un sonido que responde exactamente a la realidad no es siempre el más acertado en un videojuego. Muchas veces se exageran las cualidades de un sonido para producir el efecto deseado en el jugador.

Nuestro cerebro es capaz de identificar elementos presentes en el videojuego y asociarlos a épocas históricas, películas o experiencias que hacen más verosímiles ciertos tipos de música que otros en cada tipo de juego según su ambientación. El objetivo es, por tanto, crear audio creíble por el jugador, ya que dará un valor añadido a la capacidad de inmersión del videojuego.

Tipos y estilos de música

En un videojuego, el tipo de música se diferencia por el momento o parte del juego en la que suena. Puede ir desde la necesaria para una escena cinemática, un tema principal para el título, la música ambiental que suena durante el juego, escenas cinemáticas, eventos especiales, etc. El estilo de música depende de las necesidades del videojuego, siendo muy común encontrarse en la necesidad de componer muy diferentes estilos para un mismo proyecto.

Stingers (fanfarrias)

Son piezas musicales de pocos segundos que dan respuesta a acciones concretas del jugador para llamar su atención. Para mantener la ambientación musical del juego, estas piezas se componen con las mismas características e instrumentación que la música de fondo, pero son mucho más llamativas y sonoras.

Composición

Aunque no hay una receta mágica que asegure un buen resultado cuando se trabaja en la música de un videojuego, pueden seguirse una serie de buenas prácticas que harán el trabajo más fácil.

Crear una paleta de sonidos, escuchar música de otros autores en el estilo que pide el videojuego, guardar ideas sueltas que puedan ser útiles en un momento de poca creatividad, buscar inspiración en librerías de sonidos, experimentar con combinaciones de sonidos o configuración que no hayamos probado antes o, cuando sea posible, jugar a una versión de prueba del videojuego en el que trabajemos, pueden ser formas de agilizar el trabajo y estimular la creatividad para encontrar la inspiración necesaria.

Figura 30.4: El estilo orquestal es uno de los estilos de música más utilizados hoy día en las grandes producciones, muchas veces grabado con orquestas reales cuando se dispone de suficiente presupuesto.

Coordinando el trabajo

Una tarea a realizar antes y durante la creación de efectos sonoros es mantener un contacto constante con los desarrolladores y tratar de recopilar la mayor información posible que nos pueda ayudar en nuestro trabajo.

Seleccionar una serie de sonidos (la paleta de sonidos) adecuados a la ambientación y dinámica del videojuego en el que estemos trabajando, además de facilitarnos el trabajo de composición y evitarnos perder tiempo buscando un sonido concreto, nos llevará a crear una banda sonora coherente, conexa y con una personalidad constante durante el juego.

Creación de efectos sonoros

Hasta que el diseño y programación no están lo suficientemente avanzados, es probable que el equipo desarrollador no sepa concretar cuántos sonidos y con qué características concretas son necesarios en el videojuego. Cuanto más grande sea el proyecto, más difícil será concretar la lista de sonidos necesarios, por lo que el proceso de creación de efectos sonoros debe aplazarse lo máximo posible en la planificación del proyecto.

Antes de comenzar a trabajar, debemos tener claro el **género del videojuego**, la calidad sonora y el tamaño máximo de los archivos de sonido, qué motor de sonido se va a utilizar y si procesará los sonidos en tiempo real, si debemos crear sonidos ambientales y cuál debe ser su duración, qué prioridad tienen unos sonidos sobre otros, si el juego tendrá diálogos y en qué idiomas, cuál será el equipo de reproducción típico que tendrá el jugador del videojuego, si los efectos sonoros sonarán sobre música de fondo, si hay librerías de sonido a nuestra disposición y cuáles son, qué nombres deben tener los ficheros a entregar o qué reglas de nombrado siguen y si podemos trabajar con una versión de prueba de videojuego.

Una buena forma de empezar a trabajar es crear una **paleta de sonidos**. Esta tarea consiste en buscar unos cien o doscientos sonidos que tengamos previamente en nuestro ordenador, procedentes de librerías de sonidos o que hayamos grabado previamente para otros videojuegos y que sean del estilo del videojuego en el que vayamos a trabajar.

Debemos tratar de que estos sonidos peguen entre sí y que tengan una calidad similar. En caso de no disponer de librerías de sonidos o de trabajo anterior, debemos obtener estos sonidos por otros medios: grabándolos nosotros mismos, adquiriendo alguna de las múltiples librerías de efectos sonoros que existen, o generándolos a partir de instrumentos virtuales o samplers.

Esta paleta de sonidos no tiene por qué contener necesariamente los sonidos que elegiremos para el videojuego, sino que serán la base a partir de la cual trabajaremos. Superponer (un software multipistas facilitará esta tarea), mezclar, procesar, combinar, cambiar de tono, texturizar, reorganizar, cortar, dar la vuelta, invertir, comprimir o expandir son técnicas que podemos utilizar de diferentes formas sobre todos estos sonidos base y que nos permitirán muy probablemente ser capaces de crear los sonidos que se nos haya pedido. En caso de encontrar nuevos tipos de sonido que no podamos conseguir con las anteriores técnicas a partir de nuestra paleta de sonidos, tendremos que crearlos desde cero.

No existen fórmulas mágicas para crear efectos de sonido: combinando las técnicas mencionadas con sonidos de la vida real grabados de diferentes formas o generados con sintetizadores o samplers se pueden conseguir cualquier sonido imaginable. Combinar sonidos que a priori pueden parecer no tener nada que ver y experimentar con cualquier idea son también prácticas claves en la creación de efectos sonoros.

La utilización de librerías de sonidos, aunque sea muy tentador para los desarrolladores, casi nunca es la mejor opción. Si nos fijamos, en muchas películas, anuncios, trailers y videojuegos, podemos encontrar los mismos sonidos repetidos una y otra vez. Crear sonidos originales para un videojuego da la exclusividad y adaptación necesaria que el jugador busca y agradece.

30.1.5. Programación de audio

Un videojuego es un programa, normalmente grande y complejo. Parte de este programa debe estar dedicada a ejecutar las órdenes necesarias para que el ordenador, videoconsola, teléfono móvil o plataforma en la que se ejecute el videojuego reproduzca en el momento necesario los sonidos y música creados por el compositor.

El sonido, por lo general, no es la parte a la que se presta mayor atención en un videojuego. Durante años, al audio se le ha dado aproximadamente un 10 % del total de memoria, espacio, procesado, personal, presupuesto y publicidad en los videojuegos. Los **primeros videojuegos** se ejecutaban en máquinas en las que la mayor parte de recursos se dedicaban a la presentación de imágenes en la pantalla. Esto ha sido una tónica común en la evolución de las plataformas en las que se ejecutan los videojuegos, pero según han avanzado tecnológicamente estas plataformas, y según se han ido mejorando su capacidad de vídeo, más protagonismo ha ido tomando el sonido.

En lo que se refiere a la **programación del sonido**, la evolución ha sido lenta y tortuosa hasta llegar a las técnicas y posibilidades actuales. En un principio, las herramientas de desarrollo o bien no existían o eran particulares de la plataforma para la que se desarrollaba. Los programas que reproducían música y sonidos se desarrollaban en lenguaje ensamblador y compiladores específicos de la plataforma, y los músicos estaban limitados enormemente tanto por la propia plataforma como por su habilidad como programadores. Todo ello, a causa de la íntima relación que existía entre el contenido y el programa.

Con el tiempo, y paralelamente a la evolución tecnológica de las plataformas, las herramientas para crear contenido fueron madurando. Esto permitió la separación progresiva de contenido y software propiamente dicho, y comenzaron a aparecer motores y editores de contenido por separado.

Hoy en día, y con la complejidad que han adquirido los videojuegos, es posible comprar motores software de todo tipo para crear videojuegos, permitiendo a grupos de desarrolladores centrarse en programar las partes del videojuego que lo diferenciarían de otros delegando en estos motores para el resto de partes, y a los creadores de contenido dedicarse únicamente a su labor: crear música y sonidos sin tener que ser expertos en programación.

Uno de los mayores problemas con el que se encuentran los equipos desarrollo de videojuegos a la hora de crear un título multiplataforma es que no existe consistencia en las características de audio entre las múltiples plataformas de videojuegos, y esto provoca tener que desarrollar motores de sonido específicos para cada una de ellas por separado. En cierta medida, la utilización de APIs de terceros puede solucionar este problema.

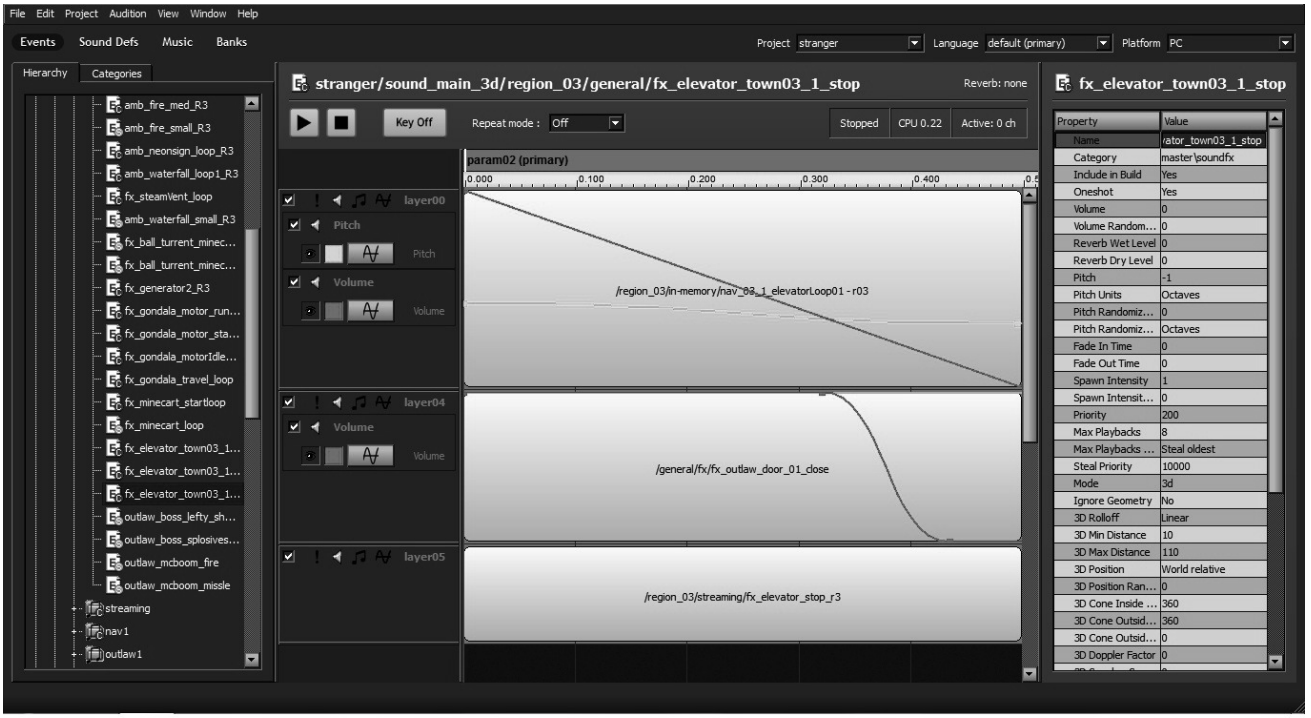


Figura 30.5: Algunos motores y APIs de audio vienen acompañadas de herramientas para la creación y organización del material sonoro de un videojuego. En la imagen, FMOD Designer.

Motores y APIs

En un videojuego, el sonido debe adaptarse a las acciones y decisiones que tome el jugador. Idealmente, el **motor de audio** de un videojuego debería ser capaz de reproducir, pausar y parar un sonido en cualquier instante, cambiar su volumen y panorama (lugar estereofónico que ocupa), reproducirlo indefinidamente, saltar a otro punto del sonido o música durante su reproducción o pasar de una pieza musical a otra mediante una transición.

Dado que un videojuego puede desarrollarse para varias plataformas, debe minimizarse la creación de contenido e implementaciones específicas para cada plataforma, proporcionando ciertos comportamientos consistentes en todas ellas.

Los motores y APIs de audio permiten a los desarrolladores no perder tiempo en programar código específico para diferentes plataformas y a los creadores de contenido ser capaces de familiarizarse con el proceso de integración del sonido en los videojuegos. Todo ello, con el objetivo final de aumentar la productividad, la eficiencia y la calidad del producto final.

Los programadores encargados del audio pueden elegir utilizar una de las diferentes **herramientas software** que existen en el mercado para facilitar su tarea o programar desde cero un motor de sonido. Esta decisión debe tomarse en las primeras fases del ciclo de creación del videojuego, para que en el momento en el que el desarrollador o grupo de desarrolladores especializados en audio que llevarán a cabo esta labor sepan a qué tipo de reto deben enfrentarse.

Desde el punto de vista del creador de contenido, esta información también resulta relevante, ya que en multitud de ocasiones serán estas APIs las encargadas de reproducir este contenido, y porque algunas de ellas vienen acompañadas de herramientas que debe conocer y ser capaz de manejar.

Existe un amplio número de **subsistemas de audio** en el mercado escritos explícitamente para satisfacer las necesidades de una aplicación multimedia o un videojuego. Estos sistemas se incorporan en librerías que pueden asociarse al videojuego tanto estática como dinámicamente en tiempo de ejecución, facilitando la incorporación de un sistema de audio completo al programador sin conocimientos especializados en este ámbito.

La facilidad de integración de estas librerías varía según su desarrollador y la plataforma a la que van dirigidas, por lo que siempre existirá una cierta cantidad de trabajo a realizar en forma de código que recubra estas librerías para unir el videojuego con la funcionalidad que ofrecen.

Algunas de las **librerías más utilizadas** hoy día son OpenAL, FMOD, Miles Sound System, ISACT, Wwise, XACT, Beatnik Audio Engine o Unreal 3 Sound System.

30.2. Video digital en los videojuegos

A lo largo de los capítulos y módulos previos, han quedado pantentes algunos de los importantes avances realizados en el marco de los gráficos 3D en los últimos años.

Por su parte, los avances en el ámbito del vídeo digital no han sido menos. Así, estamos acostumbrados a reproducir videos en nuestros dispositivos portátiles de mp4, ver video a través de flujos (del inglés *streamming*) bajo demanda por internet, tener televisión en alta definición y reprodutores de DVDs y discos Blue-Ray en el salón de nuestros hogares, etc. Es decir, el video sea o no de alta definición (del inglés *High Definition*, HD), es algo común en nuestros días.

El mundo de los videojuegos puede aprovecharse del efecto sinérgico conseguido mediante la integración de las tecnologías de video digital junto a las de gráficos 3D. Con ello, a lo largo del resto del capítulo, se verá como pueden enriquecerse escenarios 3D mediante la integración de video digital.

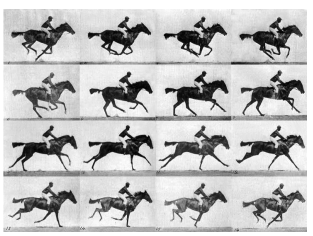


Figura 30.6: Imagenes de la serie de fotografías de Muybridge. A lo largo de la pista, se dispusieron 16 cámaras de alta velocidad. Al pasar ante cada una ellas, el caballo rompía el hilo que la accionaba provocando el disparo. La sucesión de cada fotograma permite generar el movimiento.

30.3. Gráficos 3D y video digital

Los gráficos 3D y el video digital son tecnologías complementarias, y un buen ejemplo de ello son los Videojuegos. Con gráficos en 3D, se pueden crear mundos y personajes imaginarios que nunca existieron, e incluso interactuar con ellos. Sin embargo, en ocasiones es difícil que estos gráficos sean lo suficientemente realistas. Es en este punto donde el video digital entra en escena. Aunque el video no sea interactivo, es capaz de captar el mundo real con gran cantidad de detalles e incluso con características tridimensionales, como ha quedado patente en numerosas producciones cinematográficas recientes.

Ambas tecnologías tienen mucho en común. En los gráficos 3D los objetos son matemáticamente transformados a coordenadas 2D para su visualización en la pantalla. Las texturas de las superficies de los objetos están almacenados en imagenes (mapas de bist) que son adheridas a los objetos.

Cuando se desea mostrar un video por pantalla, un área determinada de la pantalla se designa para mostrar el vídeo, ya sea en un rectángulo dentro de una ventana o a pantalla completa. Este video está formado por fotogramas (figura 30.6), y cada fotograma es un mapa de bits que se *"estira"* para adaptarse al rectángulo de destino.

Como puede verse, la principal diferencia entre gráficos 3D y video digital, radica en que el mapa de bits es actualizado varias veces por segundo en el caso del video, mientras que la mayoría de los modelos 3D usan texturas estáticas.

En las próximas secciones se mostrarán algunos conceptos esenciales necesarios para entender los formatos de video digital para posteriormente pasar a ver cómo puede realizarse su integración dentro de escenarios creados mediante Ogre 3D.

30.4. Estándares en video digital

En este capítulo nos centraremos en la integración del video digital junto a gráficos 3D. El video digital es aquel que emplea una representación digital –en lugar de analógica– de la señal de video.

Al hablar de video digital, es conveniente destacar dos factores clave que intervienen directamente sobre la calidad del vídeo: la captura y el almacenamiento. La captura de vídeo está fuera del alcance de este capítulo. En lo que respecta al almacenamiento, el video digital puede grabarse directamente sobre soportes digitales –como DVDs, discos duros o memorias flash– o bien sobre soportes analógicos –como cintas magnéticas– y posteriormente distribuirse en soportes digitales.

Para la edición de video digital, al principio se requería digitalizar una fuente de vídeo analógica a un formato digital que pudiera manipularse por ordenador. Posteriormente, el empleo de los formatos digitales para cinta conocidos como DV y miniDV, posibilitó el hecho de grabar directamente en formato digital simplificando así el proceso de edición.

Figura 30.7: Quicktime fue el primer formato de video digital para ordenadores de mercano no profesional en hogares

El primer formato de video digital para el ordenador orientado al mercado no profesional en hogares fue introducido por Apple Computer en 1991. Nos referimos al formato Quicktime. A partír de este punto, el vídeo digital no sólo fue mejorando rápidamente en calidad, sino en facilidad de procesamiento y manipulación gracias a la introducción de estándares como MPEG-1, MPEG-2 y MPEG-4. Tal fue el éxito de estos estándares que fueron rápidamente adoptados por industrias como la televisión (tanto digital como satélite), soportes como DVD, la distribución de video on-line y *streaming* bajo demanda, etc.

Los principales estándares de codificación de audio y video y su uso fundamental son los que se enumeran a continuación:

Figura 30.8: DVD Video emplea MPEG-2

- **MPEG (Moving Picture Experts Group)-1:** Es el nombre de un grupo de estándares desarrollado por el grupo de expertos en imágenes en movimiento (del inglés *Moving Pictures Experts Group* MPEG) de ISO/IEC. Para el audio, el MPEG definió el MPEG-1 *Audio Layer 3*, más conocido como MP3. Es el estandar usado en el soporte de disco Video CD y en los reproductores MP3.

- **MPEG-2:** Es la designación para el grupo de estándares publicados como ISO 13818. MPEG-2 es usado para codificar audio y vídeo en señales de Televisión Digital Terrestre (TDT (Televisión Digital Terrestre) o DVB (Digital Video Broadcasting) de sus siglas en inglés *Digital Video Broadcasting*), por Satélite o Cable. Es también el formato de codificación empleado para los soportes de discos SVCD y DVD comerciales de películas y el estándar actual de las transmisiones en televisión de alta definición (HDTV (High Definition Television)).
- **MPEG-4:** Aún en desarrollo, conforma el estándar ISO/IEC 14496. Se trata del heredero de muchas de las características de MPEG-1, MPEG-2 y estándares como soporte de VRML (Virtual Reality Modeling Language) para renderizado 3D, soporte para la gestión digital de derechos externa y variados tipos de interactividad. Es el implementado por algunos *codecs* populares como DivX, Xvid, Nero Digital y Quicktime –en sus versiones 6 y 7– y en vídeo de alta definición como Blu-ray. Los principales usos de este estándar son los *streaming* en medios audiovisuales como los canales de video on-line bajo demanda, la grabación y distribución de video en memorias flash, la transmisión bidireccional como videoconferencia y emisión de televisión.
- **Ogg:** Surgido como alternativa libre al MPEG-4, se encuentra estandarizado pero aún en desarrollo por la Fundación Xiph.org. Así, permite trabajar con audio y vídeo sin tener que emplear formatos propietarios o de pago. Ogg emplea Theora para implementar la capa de vídeo y Vorbis, Speex, FLAC u OggPCM para la capa de audio. Al igual que MPEG-4, es empleado principalmente en *streaming* de medios audiovisuales y video bajo demanda.

Figura 30.9: Blu-Ray emplea MPEG-4 Part 10

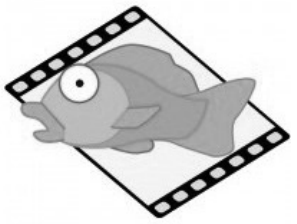


Figura 30.10: Ogg emplea Vorbis para audio y Theora para video

Cuando se desarrollan aplicaciones que deben procesar vídeo digital ya sea para su creación, edición o reproducción, debe tenerse en cuenta cuál es el estándar que se está siguiendo, dado que de ello depende el formato de archivo contenedor que se manipulará.

Cuando se habla de *formato contenedor multimedia* o simplemente formato contenedor, se está haciendo referencia a un formato de archivo que puede contener varios tipos diferentes de datos que han sido codificados y comprimidos mediante lo que se conocen como *códecs*. Los ficheros que están codificados siguiendo un formato contenedor son conocidos como *contenedores*. Éstos almacenan todos aquellos elementos necesarios para la reproducción, como el audio, el vídeo, los subtítulos, los capítulos, etc., así como los elementos de sincronización entre ellos. Algunos de los formatos contenedores más conocidos son AVI, MOV, MP4, Ogg o Matroska.

Dada la complejidad que implicaría elaborar una aplicación que interprete los ficheros contenedores así como los datos multimedia contenidos en ellos y codificados mediante *codecs*, para poder manipular el vídeo de modo que pueda ser integrado en cualquier aplicación –en nuestro caso junto a gráficos 3D–, suelen emplearse librerías o APIs (*Application Programming Interface*) que nos faciliten la tarea. En las siguientes secciones nos adentraremos en algunas de ellas.

30.5. *Plugins* de vídeo para Ogre

La librería de Ogre no proporciona manipulación de vídeo. Sin embargo, existen dos *plugins* que permiten el uso de vídeo en texturas de Ogre. El primero de ellos es Theora, el cual funciona bajo sistemas Win32 y Linux y permite reproducir ficheros Ogg. El segundo es Directshow, que sólo funciona bajo sistemas Win32 y permite reproducir cualquier vídeo de Windows.

30.5.1. Instalación de TheoraVideoPlugin

En primer lugar se deberán instalar las librerías Ogg, Vorbis y Theora. Pueden descargarse de `http://xiph.org/downloads/` o podemos emplear los repositorios de paquetes del sistema y usar comandos como:

```
apt-get install libogg-dev libvorbis-dev libtheora-dev
```

Además es preciso instalar la librería *C++ Portable Types* (ptypes) descargable de `http://www.melikyan.com/ptypes/`. Una vez descargado el *tarball*, puede descomprimirse y compilarse con la siguiente secuencia de comandos:

```
tar zxvf ptypes-2.1.1.tar.gz
cd ptypes-2.1.1
make
sudo make install
```

Finalmente ya se puede instalar el plugin de Theora disponible en `http://ogrevideo.svn.sf.net/`. Una vez descargado el código, bien empleando subversión o bien descargando directamente el *tarball* o fichero .tar.gz, en el directorio *trunk* ejecutamos la siguiente secuencia de comandos:

```
./autogen.sh
export CXXFLAGS=-I/usr/include/OGRE/
./configure
make
sudo make install
ln -s /usr/local/lib/Plugin_TheoraVideoSystem.so \
/usr/lib/OGRE/Plugin_TheoraVideoSystem.so
```

30.5.2. Incluyendo vídeo en texturas

Para emplear vídeo en texturas, bastará con incluir en el bloque *texture_unit* un bloque *texture_source ogg_video*, para indicar que se desea introducir el vídeo codificado mediante Theora en un contenedor ogg. Este bloque permite definir:

- *filename*: nombre del fichero. Este parámetro es imprescindible.
- *play_mode*: establece el estado *play/pause* de comienzo. Por defecto es *play*.
- *precache*: cuántos frames debe renderizar de antemano. Por defecto es 16.
- *output*: modo de salida (rgb, yuv, grey).

En el ejemplo de la figura 30.11 se define un material *SimpleVideo* donde en su textura se ha definido un *texture_source ogg_video* con un fichero contenedor ogg indicado en el parámetro *filename*.

```
material VideoMaterial {
  technique {
    pass {
      texture_unit {
        texture_source ogg_video {
          filename clip.ogg
          precache 16
          play_mode play
        }
      }
    }
  }
}
```

Figura 30.11: Material que incluye textura con vídeo.

30.6. Reproducción de vídeo con GStreamer

El uso de plugins de Ogre puede llevarnos a limitaciones a la hora de reproducir vídeo junto a gráficos 3D. Existe una alternativa a ellos: el empleo de entornos de desarrollo de vídeo digital especializados.

En esta sección se mostrará uno de ellos, concretamente GStreamer, y se verá cómo puede integrarse dentro de nuestros gráficos 3D con Ogre.

30.6.1. Instalación del *framework* de desarrollo y librerías necesarias

Antes de comenzar a trabajar con GStreamer deberán instalarse el *framework* y las librerías en el sistema. Para ello, en sistemas basados en Debian bastará con ejecutar indicarle a *apt-get*:

```
# apt-get install gstreamer0.10-alsa gstreamer0.10-plugins-base \
   gstreamer0.10-plugins-good libgstreamer0.10-dev \
   libstreamer-plugins-base0.10-dev
```

Si no se dispone de un sistema de gestión de paquetes, podrán descargarse de la dirección `http://gstreamer.freedesktop.org/`.

A la hora de compilar las aplicaciones, deberán añadirse a los flags de compilación aquellos devueltos por:

```
# pkg-config --cflags gstreamer-0.10
```

para la compilación y

```
# pkg-config --libs gstreamer-0.10
```

para el enlazado.

30.6.2. Introducción a GStreamer

GStreamer es un entorno de desarrollo (*framework*) para la creación de editores y reproductores de vídeo, aplicaciones de emisión de *streamming* de vídeo, etc. Está basado en el *pipeline* de vídeo del OGI (Oregon Graduate Institute) y en DirectShow, por lo que muchos de los conceptos que se verán en esta sección podrán extrapolarse a éste último.

GStreamer puede conectarse a otros *frameworks* multimedia con el fin de permitir reutilizar *codecs*, además de emplear los mecanismos de entrada-salida de cada plataforma. Así, usa OpenMAX-IL mediante gst-openmax para funcionar en Linux/Unix, DirectShow para plataformas Windows y QuickTime para Mac OS X.

La principal característica de GStreamer es que está constituido por varios *codecs* y funcionalidades que pueden ser cargados a modo de *plugins*. Del mismo modo, puede extenderse por medio de nuevos *plugins*. Esto le permite poder trabajar con multitud de formatos contenedores, protocolos de *streaming* bajo demanda, *codecs* de vídeo y audio, etc.

Los *plugins* de GStreamer pueden clasificarse en:

- gestores de protocolos
- fuentes de audio y vídeo
- manejadores de formatos con sus correspondientes *parsers*, formateadores, *muxers*, *demuxers*, metadatos, subtítulos, etc.
- *codecs*, tanto de codificación como de decodifiación
- filtros, como conversores, mezcladores, efectos, etc.
- *sinks* para volcar la salida de audio y vídeo resultante de todo el proceso

En la figura 30.12 se muestra un esquema de la arquitectura de GStreamer. En la parte superior de la figura puede verse cómo tanto las herramientas de GStreamer como otras aplicaciones multimedia como reproductores de video y audio, herramientas de VoIP y video/audio conferencia, servidores de *streaming* de audio y/o video, etc., emplean el *framework* de GStreamer. Dicho *framework* tiene a su disposición un conjunto de *plugins* (gestores de protocolos, fuentes de audi y video, manejadores de formatos, etc.) tal y como se muestra en la parte inferior de la figura. Por su parte, el *framework* de GStreamer consta de una arquitectura de en *pipeline* o flujos de información. Los *plugins* pueden entrelazarse e interactuar por medio de estos *pipelines*, de manera que es posible escribir complejas aplicaciones para la gestión de audio y vídeo.

La herramienta en línea de comandos *gst-launch* resulta muy útil para realizar algunas pruebas y prototípos rápidos de construcción de *pipelines*. Veamos algunos ejemplos.

Playbin2 es un *plugin* de GStreamer que proporciona un reproductor tanto de audio como de vídeo. Así, un comando sencillo para la reproducción de un fichero en la correspondiente ventana de *playbin2* es:

```
# gst-launch playbin2 uri=''file:///path/to/file.ogg''
```

El siguiente comando produce la misma salida, pero especifica toda la *pipe*, en lugar de dejarlo en manos del reproductor *playbin2*. Fíjese cómo ha creado un *demux* de ogg al que se le ha asignado un nombre. A continuación su salida se pasa tanto a una cola para el decodificador de Theora para el vídeo, como a un decodificador de Vorbis para el audio:

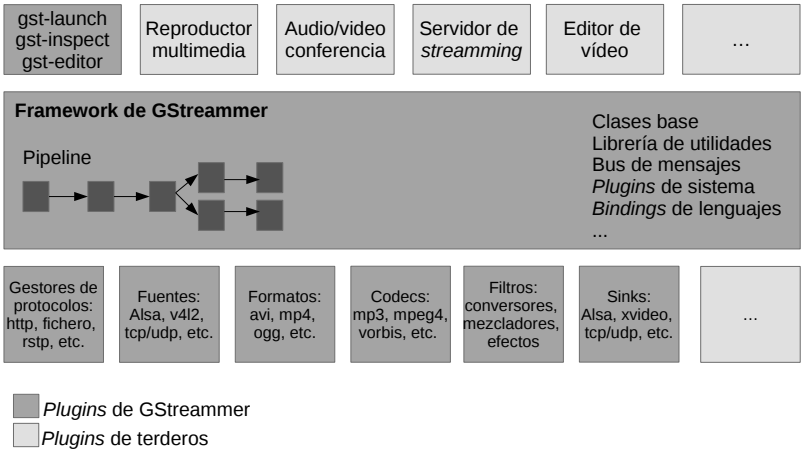


Figura 30.12: Arquitectura de GStreamer.

```
# gst-launch filesrc location=''/path/to/file.ogg'' \
   ! oggdemux name="demux"  \
   demux. ! queue ! theoradec ! xvimagesink   \
   demux. ! vorbisdec ! audioconvert ! pulsesink
```

Para trabajar con la cámara web se empleará *v4l2* para capturar frames de la cámara. Un ejemplo es el siguiente comando donde se captura un frame desde *v4l2src*, se procesa por *ffmpegcolorspace* para que no aparezca con colores "extraños", transforma a formato png con *pngenc* y se pasa a un fichero llamado *picture.png* con *filesink*

```
 # gst-launch-0.10 v4l2src ! ffmpegcolorspace ! pngenc \
   ! filesink location=picture.png
```

Si lo que queremos es mostrar todos los frames que recoga la cámara web directamente en una ventana, podemos hacerlo mediante el siguiente *pilepile*:

```
# gst-launch-0.10 v4l2src ! ffmpegcolorspace ! ximagesink
```

30.6.3. Algunos conceptos básicos

Antes de adentrarnos en la programación de aplicaciones con GStreamer es preciso describir algunos conceptos básicos.

- **Elementos.** Son los objetos más importantes en GStreamer. Las aplicaciones que empleen GStreamer especificarán cadenas de elementos unidos unos a otros para construir el *pipeline*. Los elementos harán fluir la información a través de la cadena. Cada elemento tiene una función concreta y bien definida, que puede ser leer datos de un fichero, decodificar los datos o poner los datos en la tarjeta de vídeo.

- **Pads.** Son las conexiones de entradas y salidas de los elementos. Cada *pad* gestiona datos de tipo específico, es decir, restringen el tipo de datos que fluye a través de ellos. Son el elemento que permiten crear las *pipelines* entre elementos.
- **Bins y Pipelines.** Un *bin* es un contenedor de elementos. Una *pipeline* es un tipo especial de *bin* que permite la ejecución de todos los elementos que contiene. Un *bin* es un elemento, y como tal puede ser conectado a otros elementos para construir una *pipeline*.
- **Comunicación.** Como mecanismos de comunicación GStreamer proporciona:
  - **Buffers.** Objetos para pasar datos entre elementos de la *pipeline*. Los buffers siempre van desde los fuentes hasta los *sinks*
  - **Eventos.** Objetos enviados entre elementos o desde la aplicación a los elementos.
  - **Mensajes.** Objetos enviados por los elementos al bus de la *pipeline*. Las aplicaciones pueden recoger estos mensajes que suelen almacenar información como errores, marcas, cambios de estado, etc.
  - **Consultas.** Permiten a las aplicaciones realizar consultas como la duración o la posición de la reproducción actual.

30.6.4. GStreamer en Ogre

Una vez introducidos los principios de funcionamiento de GStreamer, así como los objetos que maneja, se pasará a mostrar cómo puede integrarse en Ogre. Para ello se irá desgranando un ejemplo en el que se usa *playbin2* de GStreamer para reproducción de vídeo en una textura Ogre, que será mostrada mediante el correspondiente *overlay*. Comenzaremos escribiendo el método *createScene*.

Listado 30.1: Método createScene para cargar vídeo en un overlay

```
1  void GStreamerPlayer::createScene(){
2    // Establecer cámara, luces y demás objetos necesarios
3    ...
4
5    mVideoTexture = Ogre::TextureManager::getSingleton().createManual
         (
6                    "VideoTexture",
7            ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
8            TEX_TYPE_2D,
9            1, 1,
10           0, PF_B8G8R8A8,
11           TU_DYNAMIC_WRITE_ONLY);
12
13   mVideoMaterial = MaterialManager::getSingleton().create(
14            "VideoMaterial",
15            ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
16   Ogre::Technique* technique = mVideoMaterial->createTechnique();
17   technique->createPass();
18   mVideoMaterial->getTechnique(0)->getPass(0)->
19     createTextureUnitState(mVideoTexture->getName());
20
21   mVideoOverlay = OverlayManager::getSingleton().create("overlay");
22   OverlayContainer* videoPanel =
23     static_cast<OverlayContainer*>(OverlayManager::getSingleton().
24                    createOverlayElement("Panel", "videoPanel"));
25
26   mVideoOverlay->add2D(videoPanel);
27   mVideoOverlay->show();
28   videoPanel->setMaterialName(mVideoMaterial->getName());
```

```
29
30   gst_init(0, 0);
31
32   if (!g_thread_supported()){
33     g_thread_init(0);
34   }
35
36   mPlayer = gst_element_factory_make("playbin2", "play");
37
38   GstBus* bus = gst_pipeline_get_bus(GST_PIPELINE(mPlayer));
39   gst_bus_add_watch(bus, onBusMessage, getUserData(mPlayer));
40   gst_object_unref(bus);
41
42   mAppSink = gst_element_factory_make("appsink", "app_sink");
43
44   g_object_set(G_OBJECT(mAppSink), "emit-signals", true, NULL);
45   g_object_set(G_OBJECT(mAppSink), "max-buffers", 1, NULL);
46   g_object_set(G_OBJECT(mAppSink), "drop", true, NULL);
47
48   g_signal_connect(G_OBJECT(mAppSink), "new-buffer",
49            G_CALLBACK(onNewBuffer), this);
50
51   GstCaps* caps = gst_caps_new_simple("video/x-raw-rgb", 0);
52   GstElement* rgbFilter = gst_element_factory_make("capsfilter",
53                             "rgb_filter");
54   g_object_set(G_OBJECT(rgbFilter), "caps", caps, NULL);
55   gst_caps_unref(caps);
56
57   GstElement* appBin = gst_bin_new("app_bin");
58
59   gst_bin_add(GST_BIN(appBin), rgbFilter);
60   GstPad* rgbSinkPad = gst_element_get_static_pad(rgbFilter,
61                              "sink");
62   GstPad* ghostPad = gst_ghost_pad_new("app_bin_sink", rgbSinkPad);
63   gst_object_unref(rgbSinkPad);
64   gst_element_add_pad(appBin, ghostPad);
65
66   gst_bin_add_many(GST_BIN(appBin), mAppSink, NULL);
67   gst_element_link_many(rgbFilter, mAppSink, NULL);
68
69   g_object_set(G_OBJECT(mPlayer), "video-sink", appBin, NULL);
70   g_object_set(G_OBJECT(mPlayer), "uri", uri, NULL);
71
72   gst_element_set_state(GST_ELEMENT(mPlayer), GST_STATE_PLAYING);
73 }
```

1. **Inicialización de cámara, luces, etc.** Como siempre, al comienzo del método se inicializará la cámara, las luces y en general todo aquello que sea necesario para construir la escena.

2. **Creación de textura, material y overlay.** En el método *createScene* se deberán crear la textura, el material al que aplicar el *overlay*, y el overlay en el que se reproducirá el vídeo. Todo esto puede verse en las líneas `5-11`, `13-19` y `21-28` respectivamente.

3. **Inicialización de GStreamer y soporte multihilo.** A continuación, dado que toda aplicación que emplee las librerías de GStreamer deberán comenzar por inicializarlo, se hará uso de la función *gst_init*. La llamada a esta función inicializará la librería y procesará los parámetros específicos de GStreamer. Así, tras la creación de la textura, el material y el overlay se invocará a la función *gst_init* `30`, y se activará el soporte multihilo invocando `32-34`.

4. **Captura de mensajes.** Para poder recoger los mensajes producidos por *playbin2* desde el bus del *pipeline*, se añadirá el método que se invocará cada vez que se reciba un mensaje mediante la función *gst_bus_add_watch* aplicada sobre el bus, tal y como se muestra en las líneas `38-40`.

5. **Creación y configuración de objeto *appsink*.** El siguiente paso es el de crear un objeto *appsink* que hará que el vídeo sea pasado a la textura de Ogre (42). A continuación se hará que *appsink* emita señales de manera que nuestra aplicación conozca cuándo se ha producido un nuevo *frame* (44), hacer que el tamaño del buffer sea de uno (45), y finalmente volcarlo (46). Al final habrá que indicar cuál es la función que se encargará de procesar los *frames* que genere *appsink* mediante la función *g_signal_connect* (48).

6. **Creación de filtro *video/x-raw-rgb*.** A continuación se deberá crear un filtro que produzca datos RGB tal y como se muestra en las líneas (51-55).

7. **Creación de un *bin* que conecte el filtro con *appsink*.** Seguidamente se deberá crear un *bin* (57) al que se le añadirá el filtro y un *pad* que conecte la salida del filtro con la entrada de *appsink* (59-64). A continuación se añadira el *appsink* al *bin*.

8. **Reempazo de la ventana de salida** El paso final es el de reemplazar la ventana de salida por defecto conocida por *video-sink*, por nuestra aplicación (69) e indicar la uri del vídeo a ejecutar (70)

9. **Comienzo de la visualización del vídeo** Bastará poner el estado de nuestro objeto *playbin2* a *"play"*.

Como ha podido verse, a lo largo del código anterior se han establecido algunos *callbacks*. El primero de ellos ha sido *onBusMessage* en la línea (39) del listado anterior. En el listado siguiente es una posible implementación para dicho método, el cual se encarga de mostrar el mensaje y realizar la acción que considere oportuna según el mismo.

Listado 30.2: Método onBusMessage para captura de mensajes

```
1  gboolean GStreamerPlayer::onBusMessage(
2      GstBus* bus, GstMessage* message, gpointer userData){
3
4    GstElement* player = getPlayer(userData);
5
6    switch (GST_MESSAGE_TYPE(message)){
7    case GST_MESSAGE_EOS:
8      std::cout << "End of stream" << std::endl;
9      gst_element_set_state(GST_ELEMENT(player), GST_STATE_NULL);
10     break;
11
12   case GST_MESSAGE_ERROR:
13     std::cout << "Error" << std::endl;
14     gst_element_set_state(GST_ELEMENT(player), GST_STATE_NULL);
15     break;
16
17   default:
18     break;
19   }
20   return true;
21 }
```

El otro *callback* es *onNewBuffer* en la línea (49) del mismo listado. Este método será invocado por GStreamer cada vez que haya un nuevo *frame* disponible. Una posible implementación puede verse en el listado 30.3 continuación, donde se pone un atributo booleano llamado *mNewBufferExists* a verdadero cada vez que llega un nuevo frame.

Listado 30.3: Método onNewBuffer

```
GstFlowReturn GStreamerPlayer::onNewBuffer(GstAppSink *sink,
    gpointer userData){

  GStreamerPlayer* mediaPlayer = reinterpret_cast<GStreamerPlayer
      *>(userData);
  assert(mediaPlayer);
  mediaPlayer->mNewBufferExists = true;
  return GST_FLOW_OK;
}
```

Para hacer que Ogre actualice la textura dinámicamente, se debera construir un *Listener* que herede de *ExampleFrameListener* e implemente los métodos *frameStarted* y *frameRenderingQueued*. El método *frameRenderingQueued* es aquel desde donde se actualizará la textura dinámicamente. Una posible implementación puede ser la siguiente.

Listado 30.4: Método frameRenderingQueued y frameStarted del Listener

```
bool GStreamerPlayer::Listener::frameRenderingQueued(
                              const FrameEvent& evt){
  mMediaPlayer->updateVideo();
  return ExampleFrameListener::frameRenderingQueued(evt);
}

bool GStreamerPlayer::Listener::frameStarted(
                        const FrameEvent& evt){
  return mMediaPlayer->mRunning &&
    ExampleFrameListener::frameStarted(evt);
}
```

El método *frameRenderingQueued* encargado de actualizar la textura, invoca al método *updateVideo* del objeto *mMediaPlayer*. Una implementación es la siguiete.

Listado 30.5: Método updateVideo

```
void GStreamerPlayer::updateVideo(){
  if (mNewBufferExists){
    GstBuffer* buffer;
    g_signal_emit_by_name(mAppSink, "pull-buffer", &buffer);

    GstCaps* caps = gst_buffer_get_caps(buffer);

    int width = 0;
    int height = 0;
    int ratioNum;
    int ratioDen;
    float pixelRatio = 1.0;

    for (size_t i = 0; i < gst_caps_get_size(caps); ++i){
      GstStructure* structure = gst_caps_get_structure(caps, i);
      gst_structure_get_int(structure, "width", &width);
      gst_structure_get_int(structure, "height", &height);

      if (gst_structure_get_fraction(structure,
                    "pixel-aspect-ratio",
                    &ratioNum, &ratioDen)){
    pixelRatio = ratioNum / static_cast<float>(ratioDen);
      }
    }

    if (width && height &&
    (width != mVideoWidth || height != mVideoHeight)){
      mVideoWidth = width;
      mVideoHeight = height;
```

```
30
31        TextureManager* mgr=Ogre::TextureManager::getSingletonPtr();
32
33        if (!mVideoTexture.isNull()){
34      mgr->remove(mVideoTexture->getName());
35        }
36
37        mVideoTexture = Ogre::TextureManager::getSingleton().
38      createManual(
39                "VideoTexture",
40                ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
41                TEX_TYPE_2D,
42                mVideoWidth, mVideoHeight,
43                0, PF_B8G8R8A8,
44                TU_DYNAMIC_WRITE_ONLY);
45
46        mVideoMaterial->getTechnique(0)->getPass(0)->
47      removeAllTextureUnitStates();
48        mVideoMaterial->getTechnique(0)->getPass(0)->
49      createTextureUnitState(mVideoTexture->getName());
50
51        float widthRatio =
52      mVideoWidth / static_cast<float>(mWindow->getWidth()) *
            pixelRatio;
53        float heightRatio =
54      mVideoHeight / static_cast<float>(mWindow->getHeight());
55        float scale =
56      widthRatio > heightRatio ? widthRatio : heightRatio;
57
58        mVideoOverlay->setScale(widthRatio/scale, heightRatio/scale);
59      }
60
61      HardwarePixelBufferSharedPtr pixelBuffer =
62        mVideoTexture->getBuffer();
63      void* textureData = pixelBuffer->lock(
64                          HardwareBuffer::HBL_DISCARD);
65      memcpy(textureData, GST_BUFFER_DATA(buffer),
66         GST_BUFFER_SIZE(buffer));
67      pixelBuffer->unlock();
68
69      gst_buffer_unref(buffer);
70      mNewBufferExists = false;
71    }
72  }
```

En el anterior listado puede verse cómo lo primero que se hace en la línea [2] es comprobar si hay nuevo buffer, establecido a verdadero en el listado de *onNewBuffer*. En caso afirmativo, recoge el buffer desde la señal emitida por *appsink* [3-4]. Seguidamente, recogerá información relativa a las dimensiones del vídeo [8-24] y (en nuestro caso) si el vídeo ha cambiado de tamaño crea una nueva textura borrando la previamente existente [26-49] y calculará el factor para escalar la imagen [51-58]. A continuación bloqueará el buffer de la textura, copiará los datos del nuevo frame y desbloqueará de nuevo [61-67]. Finalmente, liberará el de la memoria reservada para los datos del buffer [69] y pondrá el atributo *mNewBufferExists* a falso [70].

30.7. Comentarios finales sobre vídeo digital

En estas secciones del capítulo se ha mostrado cómo es posible sacar provecho del efecto sinérgico entre el mundo de los gráficos 3D y el vídeo digital.

Con ello, tras una introducción a algunos conceptos fundamentales, se han analizado diferentes estándares a fin de ofrecer una panorámica que ayude a tener una mejor perspectiva a la hora de seleccionar el mejor formato, atendiendo a diversas necesidades y requisitos.

Desde el punto de vista práctico, se ha abordado la integración de vídeo digital en Ogre mediante el uso de *plugins* –más concretamente el de Theora– que permitan incrustar vídeo en texturas Ogre. Además, se ha mostrado cómo sacar partido del empleo de *frameworks* de desarrollo de aplicaciones de vídeo digital, analizando con cierta profundidad GStreamer.

Capítulo 31

Interfaces de Usuario Avanzadas

Francisco Jurado Monroy
Carlos González Morcillo

Figura 31.1: Las interfaces naturales ya no pertenecen sólo al mundo de la ciencia ficción

Figura 31.2: La visión de uno de los computadores más famosos del cine: El Terminator

Las Interfaces Naturales de Usuario (del inglés *Natural User Ingerface*) son aquellas que emplean los movimientos gestuales para permitir una interacción con el sistema sin necesidad de emplear dispositivos de entrada como el ratón, el teclado o el joystick. Así, el cuerpo, las manos o la voz del usuario se convierten en el mando que le permite interactuar con el sistema.

Este es el paradigma de interacción en el que se basan tecnologías como las pantallas capacitivas tan empleadas en telefonía movil y tabletas, sistemas de reconocimiento de voz como el implantado en algunos vehículos para permitir "hablar" al coche y no despistarnos de la conducción, o dispositivos como Kinect de Xbox que nos permite sumergirnos en mundos virtuales empleando nuestro cuerpo.

Este capítulo se centrará en la construcción de interfaces naturales que empleen movimientos gestuales mediante la aplicación de técnicas de Visión por Computador o Visión Artificial y el uso de dispositivos como el mando de la Wii de Nintendo y el Kinect de XBox, abordando temas como la identificación del movimiento, el seguimiento de objetos o el reconocimiento facial, que puedan emplearse en el diseño y desarrollo de videojuegos que sigan este paradigma de interacción.

31.1. Introducción a la Visión por Computador

Quizá, lo primero que puede sugerir el término "Visión por Computador" es que se trata de aquellas técnicas que permiten "ver a través del ordenador", es decir, ver como lo haría un ordenador. Lo cierto es que la ciencia ficción se ha encargado de darnos algunas ideas al respecto, y de algún modo no va desencaminada.

Las técnicas de Visión por Computador permiten extraer y analizar ciertas propiedades del mundo real a partir de un conjunto de imágenes obtenidas por medio de una cámara. A día de hoy, cuenta con algoritmos suficientemente probados y maduros, permitiéndo construir aplicaciones que se aprovechen de todo su potencial. Además, el abaratamiento de las cámaras digitales, por otro lado cada vez más potentes, permite obtener unos resultados cada vez más sorprendentes en este campo.

En el caso que nos ocupa, las técnicas de Visión por Computador pueden ser empleadas para el tratamiento de aquellas propiedades que permitan identificar los movimientos del usuario, dotando así a la aplicación de la capacidad de actuar en consecuencia.

31.2. Introducción a OpenCV

OpenCV es una librería de código abierto multiplataforma (GNU/Linux, Windows y Mac OS X) escrita en los lenguajes de programación C/C++ y distribuida bajo licencia BSD. Sus orígenes parten de una iniciativa de Intel Reseach con caracter libre, para ser empleada con fines comerciales, educativos y de investigación.

Fue diseñada para ser eficiente en aplicaciones de tiempo real y proveer de un *framework* de Visión por Computador que fuera sencillo de usar, permitiendo así la construcción rápida de aplicaciones de Visión por Computador que fueran potentes y robustas. Cuenta además con interfaces para otros lenguajes de programación como Python, Ruby, Java, Matlab, entre otros.

A pesar de todo, si se desea un mayor rendimiento y optimización, pueden adquirirse las librerías Intel's IPP (Integraed Performance Primitives), un conjunto de rutinas optimizadas de bajo nivel para arquitecturas Intel.

Las librerías OpenCV están compuestas por cinco modulos:

- **CV**: Contiene las funciones principales de OpenCV, tales como procesamiento de imágenes, análisis de la estructura de la imagen, detección del movimiento y rastreo de objetos, reconocimiento de patrones, etc.
- **Machine Learning**: Implementa funciones para agrupación (*clustering*), clasificación y análisis de datos.
- **CXCORE**: Define las estructuras de datos y funciones de soporte para algebra lineal, persistencia de objetos, tranformación de datos, manejo de errores, etc.
- **HighGUI**: Empleada para construcción de interfaces de usuario sencillas y muy ligeras.
- **CVAUX**: Formada por un conjunto de funciones auxiliares/experimentales de OpenCV.

Figura 31.3: Logo de OpenCV

31.2.1. Instalación de OpenCV

La librería OpenCV está disponible a través de la dirección Web `http://sourceforge.net/projects/opencvlibrary/` y pueden encontrarse documentación y ejemplos en `http://opencv.willowgarage.com/wiki/` y `http://code.opencv.org/projects/OpenCV/wiki/WikiStart`.

Para realizar la instalación de la libreria, podemos descargar el código fuente directamente del enlace anterior y proceder posteriormente a su compilación para la plataforma correspondiente, o bien podemos emplear los repositorios de paquetes del sistema. Así, en un sistema basado en Debian los comandos a ejecutar serían:

```
# apt-get install build-essential
# apt-get install libavformat-dev
# apt-get install ffmpeg
# apt-get install libcv2.1 libcvaux2.1 libhighgui2.1 opencv-doc \
libcv-dev libcvaux-dev libhighgui-dev python-opencv
```

Con esto tendremos los requisitos mínimos para poder desarrollar, compilar y ejecutar nuestras aplicaciones con OpenCV.

La distribución viene acompañada de una serie de programitas de ejemplo en diferentes lenguajes de programación. Estos pueden resultar de mucha utilidad en una primera toma de contacto. Si se ha realizado la instalación mediante el sistema de paquetes, el paquete opencv-doc es el encargado de añadir dichos ejemplos. Para poder copiarlos y compilarlos haremos:

```
# cp -r /usr/share/doc/opencv-doc/examples .
# cd examples
# cd c
# sh build_all.sh
```

31.3. Conceptos previos

Antes de adentrarnos en los conceptos del procesamiento de imagen que nos ofrece la Visión por Computador con OpenCV, es preciso introducir algunos conceptos como aclaraciones sobre las nomenclaturas de las funciones, las estructuras de datos, cómo construir sencillas interfaces de usuario para testear nuestros desarrollos, etc.

31.3.1. Nomenclatura

Como nemotécnico general para la interfaz de programación en C, puede apreciarse cómo las funciones de OpenCV comienzan con el prefijo *cv* seguido de la operación y el objeto sobre el que se aplicará. Veamos algunos ejemplos:

- cvCreateImage: función OpenCV para crear una imagen.
- cvCreateMat: función OpenCV para crear una matriz.
- cvCaptureFromAVI: función OpenCV para realizar una captura de un fichero AVI.
- cvLoadImage: función OpenCV para cargar una imagen.
- cvShowImage: función OpenCV para mostrar una imagen.
- cvConvertImage: función OpenCV para convertir una imagen.
- cvSetImageROI: función OpenCV para establecer una región de interés (ROI) sobre una imagen.
- ...

31.3.2. Interfaces ligeras con HighGUI

El módulo HighGUI, permite la construcción de interfaces de usuario multiplataforma, ligeras y de alto nivel. Estas suelen estár destinadas a probar los procesamientos que se realicen con OpenCV.

HighGUI tiene funciones para permitir a los desarrolladores interactuar con el sistema operativo, el sistema de archivos, la WebCam, construir ventanas en las que mostrar imágenes y vídeos, leer y escribir ficheros (imágenes y vídeo) desde y hacia ficheros, gestión sencilla del ratón y teclado, etc.

Ventanas

Las primitivas más interesantes y comunes en una aplicación que necesite gestión de ventanas son:

- **Crear**: *cvNamedWindow("window", CV_WINDOW_AUTOSIZE)*
- **Redimensionar**: *cvResizeWindow("window", width, heigh);*
- **Posicionar**: *cvMoveWindow("window", offset_x, offset_y)*
- **Cargar imagen**: *IplImage* img=cvLoadImage(fileName)*
- **Visualizar imagen**: *cvShowImage("window",img);*
- **liberar imagen**: *cvReleaseImage(&img)*
- **Destruir**: *cvDestroyWindow("window");*

Como puede apreciarse, ninguna de las anteriores primitivas devuelve una referencia a la ventana o recibe por parámetro una referencia. Esto es debido a que HighGUI accede a las ventanas que gestiona mediante su nombre. De esta manera, cada vez que se desee acceder a una ventana concreta, habrá que indicar su nombre. Esto evita la necesidad de emplear y gestionar estructuras adicionales por parte de los programadores.

Recibir eventos del teclado

Para el teclado no existe manejador. Tan sólo se dispone de una primitiva *cvWaitKey(interval)*. Si *interval* vale 0, la llamada es bloqueante, y se quedará esperando indefinidamente hasta que se pulse una tecla. Si *interval* tiene un valor positivo, este representará el número de milisegundos que esperará a recibir un evento del teclado antes de continuar con la ejecución.

Capturando eventos del ratón

Como en la mayoría de las librerías para la construcción de interfaces de usuario, los eventos del ratón son manipulados a través del correspondiente manejador. Un ejemplo de manejador del ratón puede verse en el siguiente código:

Listado 31.1: Manejador del ratón

```
1 void mouseHandler(int event, int x, int y, int flags, void* param){
2   switch(event){
```

```
3    case CV_EVENT_LBUTTONDOWN:
4      if(flags & CV_EVENT_FLAG_CTRLKEY)
5        printf("Pulsado boton izquierdo con CTRL presionada\n");
6      break;
7
8    case CV_EVENT_LBUTTONUP:
9      printf("Boton izquierdo liberado\n");
10     break;
11   }
12 }
```

Como puede apreciarse, el manejador recibe una serie de parámetros:

- **x, y**: Las coordenadas de los pixels en los que se produce el evento
- **event**: El tipo de evento.
  - CV_EVENT_LBUTTONDOWN, CV_EVENT_RBUTTONDOWN, CV_EVENT_MBUTTONDOWN, CV_EVENT_LBUTTONUP, CV_EVENT_RBUTTONUP, CV_EVENT_RBUTTONDBLCLK, CV_EVENT_MBUTTONDBLCLK, CV_EVENT_MOUSEMOVE, CV_EVENT_MBUTTONUP, CV_EVENT_LBUTTONDBLCLK,
- **flags**: El flag que modifica al evento.
  - CV_EVENT_FLAG_CTRLKEY, CV_EVENT_FLAG_SHIFTKEY, CV_EVENT_FLAG_ALTKEY, CV_EVENT_FLAG_LBUTTON, CV_EVENT_FLAG_RBUTTON, CV_EVENT_FLAG_MBUTTON

Para registrar el manejador y así poder recibir los eventos del ratón para una ventana concreta, deberá emplearse la función *cvSetMouseCallback("window",mouseHandler,&mouseParam);*.

Barra de progreso

Otro elemento interesante en las interfaces construidas con HighGUI son las barras de progreso o *trackbar*. Estas suelen ser muy útiles para introducir valores enteros "al vuelo" en nuestra aplicación. Por ejemplo, para posicionar la visualización de un vídeo en un punto determinado de la visualización o para establecer los valores de determinados umbrales en los procesados (brillos, tonos, saturación, matices, etc.).

El manejador del evento del *trackbar* tiene el siguiete aspecto:

Listado 31.2: Manejador del *trackbar*

```
1 void trackbarHandler(int pos)
2 {
3   printf("Trackbar en la posicion: %d\n",pos);
4 }
```

Para registrarlo bastará con invocar a la primitiva *cvCreateTrackbar("bar1", "window", &trackbarVal ,maxVal , trackbarHandler)*. Si se desea establecer u obtener la posición del *trackbar* durante un momento dado de la ejecución, se emplearán las funciones *cvGetTrackbarPos("bar1","window")* y *cvSetTrackbarPos("bar1", "window", value)*, respectivamente

31.3.3. Estructura de la imagen

Las imágenes son la piedra angular en la visión por computador. OpenCV proporciona la estructura *IplImage* heredada de la *Intel Image Processing Library*. Se encuentra definida en el fichero *cxtypes.h* y la estructura es la que se muestra en el listado:

Listado 31.3: Estructura *IplImage*

```
typedef struct _IplImage
{
    int  nSize;             /* sizeof(IplImage) */
    int  ID;                /* version (=0)*/
    int  nChannels;         /* Most of OpenCV functions support
                               1,2,3 or 4 channels */
    int  alphaChannel;      /* Ignored by OpenCV */
    int  depth;             /* Pixel depth in bits: IPL_DEPTH_8U,
                               IPL_DEPTH_8S,  IPL_DEPTH_16S,
                               IPL_DEPTH_32S, IPL_DEPTH_32F and
                               IPL_DEPTH_64F are supported.  */
    char colorModel[4];     /* Ignored by OpenCV */
    char channelSeq[4];     /* ditto */
    int  dataOrder;         /* 0 - interleaved color channels,
                               1 - separate color channels.
                               cvCreateImage can only create
                               interleaved images */
    int  origin;            /* 0 - top-left origin,
                               1 - bottom-left origin (Windows
                                   bitmaps style).*/
    int  align;             /* Alignment of image rows (4 or 8).
                  OpenCV ignores it and uses widthStep instead.*/
    int  width;             /* Image width in pixels.*/
    int  height;            /* Image height in pixels.*/
    struct _IplROI *roi;    /* Image ROI.
                  If NULL, the whole image is selected. */
    struct _IplImage *maskROI;      /* Must be NULL. */
    void  *imageId;                 /* "            " */
    struct _IplTileInfo *tileInfo;  /* "            " */
    int  imageSize;         /* Image data size in bytes
                               (==image->height*image->widthStep
                               in case of interleaved data)*/
    char *imageData;        /* Pointer to aligned image data.*/
    int  widthStep;         /* Size of aligned image row
                               in bytes.*/
    int  BorderMode[4];     /* Ignored by OpenCV.*/
    int  BorderConst[4];    /* Ditto.*/
    char *imageDataOrigin;  /* Pointer to very origin of image data
                               (not necessarily aligned) -
                               needed for correct deallocation */
}
IplImage;
```

Así, para acceder a un pixel concreto para una imagen multicanal puede emplearse el siguiente código:

Listado 31.4: Acceso a los pixels de una imagen

```
IplImage* img  = cvCreateImage(cvSize(640,480),IPL_DEPTH_32F,3);

int height     = img->height;
int width      = img->width;
int step       = img->widthStep/sizeof(float);
int channels   = img->nChannels;
float * data   = (float *)img->imageData;

data[i*step+j*channels+k] = 111;
```

Del mismo modo, para acceder a todos los pixels de una imagen, bastará con implementar un bucle anidado, tal y como se muestra en el siguiente código de ejemplo encargado de convertir una imagen a escala de grises:

Listado 31.5: Operación directa sobre pixels. Convirtiendo a escala de grises.

```
#include "highgui.h"
#include "cv.h"
#include "stdio.h"

int main(int argc, char **argv) {
    IplImage *img = cvLoadImage( argv[1], CV_LOAD_IMAGE_COLOR );

    if (!img)
        printf("Error while loading image file\n");

    int width     = img->width;
    int height    = img->height;
    int nchannels = img->nChannels;
    int step      = img->widthStep;
    uchar *data   = ( uchar* )img->imageData;

    int i, j, r, g, b, byte;
    for( i = 0 ; i < height ; i++ ) {
        for( j = 0 ; j < width ; j++ ) {
            r = data[i*step + j*nchannels + 0];
            g = data[i*step + j*nchannels + 1];
            b = data[i*step + j*nchannels + 2];

            byte = ( r + g + b ) / 3;

            data[i*step + j*nchannels + 0] = byte;
            data[i*step + j*nchannels + 1] = byte;
            data[i*step + j*nchannels + 2] = byte;
        }
    }

    cvNamedWindow( "window", CV_WINDOW_AUTOSIZE );
    cvShowImage("window",img);

    cvWaitKey(0);

    cvReleaseImage(&img);
    cvDestroyWindow("window");

    return 0;
}
```

31.3.4. Almacenes de memoria y secuencias

En algunas de las operaciones que realizan los algoritmos de OpenCV, será necesario hacer la correspondiente reserva de memoria, de manera que nuestra aplicación sea capaz de construir objetos de manera dinámica. Además, se introducirá en este apartado la "secuencia" como una estructura de datos en la que almacenar nuestros objetos dinámicos o donde recibir la salida de determinados procesamientos realizados por las funciones de OpenCV.

Almacenes de memoria: *MemStorage*

Existen funciones en OpenCV que requieren de una reserva de memoria para almacenar cálculos intermedios o de salida. Por esto, OpenCV dispone de una estructura específica a la que denomina almacén de memoria *MemStorage* para permitir la gestión de memoria dinámica.

Las funciones principales para gestión de almacenes de memoria son:

- **Reservar**:
  - *CvMemStorage* cvCreateMemStorage(int block_size = 0)*
  - *void* cvMemStorageAlloc(CvMemStorage* storage, size_t size)*
- **Liberar**: *void cvReleaseMemStorage(CvMemStorage** storage)*
- **Vaciar**: *void cvClearMemStorage(CvMemStorage* storage)*

Con esto se dispone de todo lo necesario para poder trabajar con memoria dinámica en OpenCV.

Secuencias: *CVSeq*

Un tipo de objeto que puede ser guardado en los almacenes de memoria son las secuencias. Estas son listas enlazadas de otras estructuras, de manera que se pueden crear secuencias de cualquier tipo de objeto. Por su implementación, las secuencias permiten incluso el acceso directo a cada uno de los elementos.

Algunas de las principales primitivas para su manipulación son:

- **Crear**: *CvSeq* cvCreateSeq(int seq_flags, int header_size, int elem_size, CvMemStorage* storage)*
- **Eliminar**: *void cvClearSeq(CvSeq* seq)*
- **Acceder**: *char* cvGetSeqElem(seq,index)*
- **Comprobar**: *int cvSeqElemIdx(const CvSeq* seq, const void* element, CvSeqBlock** block = NULL)*
- **Clonar**: *CvSeq* cvCloneSeq(const CvSeq* seq, CvMemStorage* storage)*
- **Ordenar**: *void cvSeqSort(CvSeq* seq, CvCmpFunc func, void* userdata = NULL)*
- **Buscar**: *char* cvSeqSearch(CvSeq* seq, const void* elem, CvCmpFunc func, int is_sorted, int* elem_idx, void* userdata = NULL)*

31.4. Primera toma de contacto con OpenCV: mostrando vídeo

A modo de una primera toma de contacto, se verá un pequeño ejemplo que muestra en una ventana el contenido de un archivo de video pasado por parámetro a la aplicación, o bien la captura de la WebCam en su ausencia.

Listado 31.6: Mostrando imágenes de una captura

```
1  #include "highgui.h"
2  #include "cv.h"
3
4  int main( int argc, char** argv ) {
5    CvCapture* capture;
6    if( argc != 2 || !(capture = cvCreateFileCapture(argv[1])) )
7      capture = cvCreateCameraCapture(0);
8
9    cvNamedWindow( "Window", CV_WINDOW_AUTOSIZE );
10
11   IplImage* frame;
12   while(1) {
13     frame = cvQueryFrame(capture);
14     if(!frame) break;
15
16     cvShowImage("Window", frame);
17
18     char c = cvWaitKey(33);
19     if(c==27) break;
20   }
21
22   cvReleaseCapture(&capture);
23   cvDestroyWindow("Window");
24
25   return 0;
26 }
```

Aclarado esto, puede que el código resulte bastante autodescriptivo. Así, las líneas 1-2 muestran la inclusión de los ficheros con las definiciones de las funciones tanto para OpenCV como para Highgui. Lo primero que hace la aplicación es una mínima comprobación de parámetros 6-7. Si se ha introducido un parámetro, se entenderá que es la ruta al fichero a visualizar y se creará una captura para extraer la información del mismo mediante la llamada a *cvCreateFileCapture*. En caso de que no se hayan introducido parámetros o el parámetro no sea un fichero de video, se realizará la captura desde la WebCam empleando la llamada a *cvCreateCameraCapture*.

A continuación, en la línea 9 se creará una ventana llamada *Window* mediante la llamada a *cvNamedWindow*. Como puede verse, esta función no devuelve ningún puntero a la estructura que permita acceder a la ventana, sino que construye una *ventana con nombre*. Así, cada vez que deseemos acceder a una ventana concreta, bastará con indicar su nombre, evitando emplear estructuras adicionales.

Los *frames* de los que consta el vídeo serán mostrados en la ventana mediante el bucle de las líneas 12-20. En él, se realiza la extracción de los *frames* desde la captura (fichero de video o cámara) y se muestran en la ventana. El bucle finalizará porque se haya extraido el último *frame* de la captura łínea14 o se haya pulsado la tecla de escape 18-19 por parte del usuario. Finalmente, se liberan las estructuras mediante las correspondientes llamadas del tipo *cvRelease** 22-23.

Como puede verse, en sólo unas pocas líneas de código se ha implementado un reproductor de vídeo empleando OpenCV. En las siguientes secciones se mostrará cómo sacarle algo de partido al potencial que nos ofrece la Visión por Computador.

31.5. Introducción a los filtros

En la mayoría de las ocasiones, al trabajar con visión por computador no nos limitaremos a "mostrar video por pantalla", sino que se querrá realizar algún tipo de procesamiento sobre las imágenes que estemos capturando.

En esta sección veremos cómo realizar unas pequeñas modificaciones sobre el ejemplo anterior, pasando una serie de filtros a cada uno de los *frames* recogidos de un vídeo, a fin de mostrar sólo la silueta de las figuras. El código que lo implementa es el del siguiente listado:

Listado 31.7: Identificando bordes y siluetas

```
1  #include "highgui.h"
2  #include "cv.h"
3
4  int main( int argc, char** argv ) {
5    CvCapture* capture;
6    if( argc != 2 || !(capture = cvCreateFileCapture(argv[1])) )
7      capture = cvCreateCameraCapture(0);
8
9    cvNamedWindow( "Window_1", CV_WINDOW_AUTOSIZE );
10   cvNamedWindow( "Window_2", CV_WINDOW_AUTOSIZE );
11
12   IplImage* frame;
13   while(1) {
14     frame = cvQueryFrame( capture );
15     if( !frame ) break;
16
17     IplImage *smooth, *edge, *out;
18     smooth = cvCreateImage(cvGetSize(frame), IPL_DEPTH_8U, 3);
19     edge   = cvCreateImage(cvGetSize(frame), IPL_DEPTH_8U, 1);
20     out    = cvCreateImage(cvGetSize(frame), IPL_DEPTH_8U, 1);
21
22     cvSmooth  (frame, smooth, CV_GAUSSIAN, 11, 11, 0, 0);
23     cvCvtColor(smooth, edge, CV_BGR2GRAY);
24     cvCanny   (edge, out, 10, 50, 3 );
25
26     cvShowImage( "Window_1", frame );
27     cvShowImage( "Window_2", out );
28
29     cvReleaseImage(&out);
30     cvReleaseImage(&edge);
31     cvReleaseImage(&smooth);
32
33     char c = cvWaitKey(33);
34     if( c == 27 ) break;
35   }
36
37   cvReleaseImage(&frame);
38   cvReleaseCapture( &capture );
39   cvDestroyWindow( "Window_1" );
40   cvDestroyWindow( "Window_2" );
41
42   return 0;
43 }
```

En las líneas 9-10 se han creado dos ventanas, de manera que se puedan apreciar los frames de entrada y de salida del ejemplo. Para almacenar las transformaciones intermedias, se crearán una serie de imágenes mediante la primitiva *cvCreateImage* 17-20, cuyo tamaño es siempre el tamaño del *frame* original.

La primitiva *cvCreateImage* será una de las que más se empleen cuando desarrollemos aplicaciones con OpenCV. Esta crea la cabecera de la imagen y reserva la memoria para los datos de la misma. Para ello toma tres parámetros: el tamaño (*size*) que a su vez se compone de alto y ancho, la profundidad (*depth*) y el número de canales (*channels*).

El proceso de filtrado está codificado en las líneas 22-24. Como puede apreciarse son tres los filtros que se le aplican:

1. *Smooth* o suavizado. El primer paso es el de aplicar a cada *frame* capturado una operación de suavizado Gaussiano 11x11. Este filtro producirá una imagen más suave a la que se le ha eliminado el posible "ruido" y cambios bruscos en el color.

2. Escala de grises. Esta transformación permite pasar la imagen de RGB a escala de grises. De ahí que la imagen creada para almacenar esta transformación intermedia tenga una profundidad de 1 en la línea (19). Este es en realidad un paso intermedio requerido para generar la entrada al algoritmo de detección de contornos.

3. *Canny* o detección de contornos. El cuál permite aplicar el algoritmo *Canny* para la identificación de contornos en una imagen. Este algoritmo toma como entrada una imagen de un solo canal (la imagen en escala de grises del paso anterior), y genera otra imagen de un solo canal en la que se almacenan las siluetas o contornos identificados.

En la figura 31.4 puede verse cuál será la salida final de la ejecución de nuestra aplicación.

Figura 31.4: Salida del ejemplo. A la izquierda la imagen antes del proceso de filtrado. A la derecha los bordes detectados tras el proceso de filtrado.

Lo que se consigue aplicando el algoritmo *Canny* es identificar aquellos pixels que conforman el "borde" de los objetos, es decir, aquellos que separan segmentos diferentes de la imagen. Se trata de pixels aislados, sin una entidad propia. En realidad, un contorno es una lista de puntos que representan una curva en una imagen.

En OpenCV un contorno se representa mediante secuencias de pixels, y la estructura de datos que la implementa es *CvSeq*. A través de esta estructura podrá recorrerse el contorno completo tan sólo siguiendo la secuencia.

Listado 31.8: Uso de *cvFindContours*

```
1  IplImage* image = /*inicializada previamente*/;
2  int thresh = 100;
3
4  IplImage* gray = cvCreateImage( cvGetSize(image), 8, 1 );
5  CvMemStorage* storage = cvCreateMemStorage(0);
6
7  cvCvtColor(image, gray, CV_BGR2GRAY);
8  cvThreshold(gray, gray, thresh, 255, CV_THRESH_BINARY);
9  CvSeq* contours = 0;
10 cvFindContours(gray, storage, &contours, sizeof(CvContour),
```

```
11           CV_RETR_TREE,
12           CV_CHAIN_APPROX_SIMPLE, cvPoint(0,0) );
13 cvZero(gray);
14 if( contours )
15     cvDrawContours(gray, contours, cvScalarAll(255),
16                 cvScalarAll(255), 100, 3, CV_AA, cvPoint(0,0) );
17 cvShowImage( "Contours", gray );
```

Un modo de ensamblar todos esos pixels aislados en contornos es mediante la aplicación de la función *cvFindContours*. Esta función calcula los contornos a partir de imágenes binarias. Así, toma como entrada imágenes creadas con *cvCanny*, *cvThreshold* o *cvAdaptiveThreshold* y devuelve todos los contornos que ha identificado y los cuales pueden ser accedidos mediante la correspondiente estructura *CvSeq*.

31.6. Detección de objetos mediante reconocimiento

A la hora de proporcionar una interacción natural, uno de los elementos que deben tenerse en cuenta es el de detectar que aquello con lo que interactua nuestro computador es un rostro, de manera que pueda actuar en consecuencia.

OpenCV reconoce (o clasifica) regiones que cumplan con determinadas características aprendidas, tales como rostros, cuerpos, etc.

Para esto se hará uso del módulo de aprendizaje automático (*Machine Learning*) que incluye OpenCV. Así, el procedimiento habitual es el de entrenar un clasificador con aquellos patrones que se deseen reconocer. Tras el entrenamiento y la validación del mismo, el clasificador estará listo para identificar los patrones para los que fue entrenado.

En las siguientes subsecciones veremos cómo realizar el reconocimiento de objetos mediante el uso de clasificadores, y cómo pueden ser empleados para proporcionar interacción con el usuario.

31.6.1. Trabajando con clasificadores

Tal y como se ha mencionado, el empleo de clasificadores implica un proceso de *"aprendizaje"* de los mismos. En dicho proceso, se *"entrena"* al clasificador con una serie de ejemplos (tanto buenos como malos) ya clasificados, y al que se le conoce comúnmente como *"conjunto de entrenamiento"*. Posteriormente se *"valida"* dicho entrenamiento analizando y contrastando la salida con un *"conjunto de validación"* tambien clasificado de antemano. Contrastar en qué medida ha clasificado los casos del conjunto de validación, dará una idea de la precisión del clasificador.

Tanto el conjunto de entrenamiento como el de validación, deben contar un número importante de elementos clasificados a fin de que el proceso de aprendizaje proporcione una precisión razonable. Esto hace que el proceso de entrenamiento suela ser costoso en tiempo, tanto en la construcción de los conjuntos para el aprendizaje como en el cómputo a realizar por los diferentes algoritmos de clasificación.

Afortunadamente, existen diferentes especificaciones para los clasificadores ya entrenados de uso más común para OpenCV, que pueden ser integrados directamente en nuestras aplicaciones. Si se ha hecho la instalación a través del sistema de gestión de paquetes de nuestro sistema, en aquellos basados en Debian podrán encontrarse en */usr/share/doc/opencv-doc/examples/haarcascades/*. Si se ha descargado directamen-

te el fichero *tarball*, una vez descomprimido, estos pueden encontrarse en el directorio *OpenCV-X.X.X/data/haarcascades*. En este directorio se pueden encontrar las especificaciones de clasificadores para reconocimiento facial, reconocimiento de ojos, nariz, boca, etc.

Analicemos un ejemplo donde se muestra el uso de un reconocedor facial para una imagen.

Listado 31.9: Aplicación de clasificadores para detectar objetos

```
1  #include <cv.h>
2  #include <highgui.h>
3
4  int main(int argc, char** argv ){
5    static CvScalar colors[] = { {{0,0,255}}  , {{0,128,255}},
6                    {{0,255,255}}, {{0,255,0}},
7                    {{255,128,0}}, {{255,255,0}},
8                    {{255,0,0}}  , {{255,0,255}} };
9
10   IplImage* img = cvLoadImage("./media/The-Beatles.jpg", 1);
11   CvMemStorage* storage = cvCreateMemStorage(0);
12   CvHaarClassifierCascade* classifier = cvLoadHaarClassifierCascade
13     ("./media/haarcascade_frontalface_alt.xml", cvGetSize(img));
14
15   cvClearMemStorage(storage);
16   CvSeq* faces = cvHaarDetectObjects
17     (img, classifier, storage, 1.1, 4, 0, cvSize( 40, 50 ), cvSize(
           40, 50 ));
18
19   CvRect* r; int i;
20   for( i = 0; i < (faces? faces->total : 0); i++ ){
21     r = (CvRect*)cvGetSeqElem(faces, i);
22     cvRectangle(img, cvPoint(r->x, r->y),
23         cvPoint(r->x + r->width, r->y + r->height),
24         colors[i%8], 3, 10, 0);
25   }
26
27   cvNamedWindow("Window", CV_WINDOW_AUTOSIZE );
28   cvShowImage("Window", img );
29   cvWaitKey(0);
30
31   cvReleaseImage(&img);
32   cvDestroyWindow("Window");
33
34   return 0;
35 }
```

En la línea 5 se definen una serie de colores que serán empleados para marcar con un rectángulo los objetos detectados. En las líneas 10-12, cargamos la imagen desde un fichero, reservamos memoria para llevar a cabo el proceso de clasificación y cargamos la especificación del clasificador.

Seguidamente, en las líneas 15-17 se limpia la memoria donde se realizará el proceso de detección, y se pone a trabajar al clasificador sobre la imagen mediante la llamada a la función *cvHaarDetectObjects*. Finalizado el proceso de detección por parte del clasificador, se obtiene una secuencia de objetos (en nuestro caso caras) almacenada en *faces* de tipo *CvSeq* (estructura de datos para almacenar secuencias en OpenCV).

Bastará con recorrer dicha secuencia e ir creando rectángulos de diferentes colores (empleando *cvRectangle*) sobre cada una de las caras identificadas. Esto es lo que se realiza en el bucle de las líneas 19-25. Finalmente se muestra la salida en una ventana y espera a que el usuario pulse cualquier tecla antes de liberar la memoria reservada 27-32.

La salida de la ejecución puede apreciarse en la figura 31.5, donde puede verse la precisión del clasificador. Incluso la cara de la chica del fondo ha sido identificada. La del hombre del fondo a la izquierda no ha sido reconocida como tal, porque no corresponde con una "cara completa" y por tanto no cumple con los patrones aprendidos.

31.6.2. Interacción con el usuario

Una vez visto cómo realizar detección de objetos con OpenCV, a nadie se le escapa que el hecho de identificar un cuerpo humano, un rostro (o cualquier otra parte del cuerpo humano), etc., puede permitir a nuestras aplicaciones realizar acciones en función de dónde se encuentren dichos objetos.

Así, imagíne que pudieramos ejecutar unas acciones u otras según la posición del usuario ante la cámara, o que atendiendo al parpadeo de éste fuera posible desencadenar la realización de unas acciones u otras en el computador. Introducir los conceptos asociados a estas cuestiones es el objeto de esta subsección.

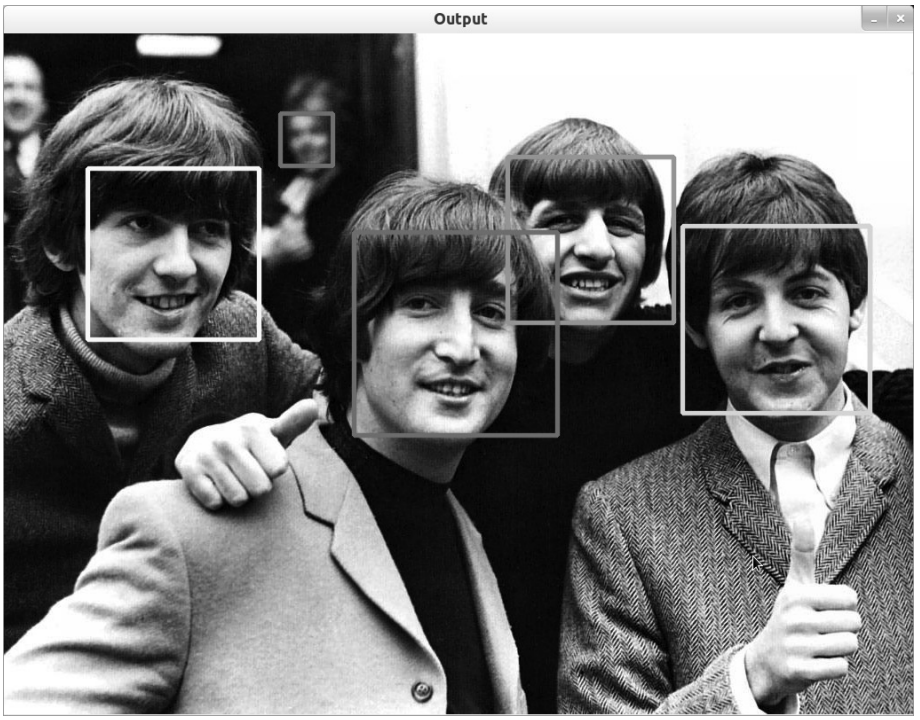


Figura 31.5: Salida del ejemplo de reconocimiento facial con OpenCV.

Veamos un sencillo ejemplo en el que se realiza la detección del rostro en los frames extraidos de un fichero de video pasádo por parámetro a la aplicación, o de la WebCam en ausencia de este parámetro. En función de dónde se encuentre el rostro, se mostrará un mensaje identificando en qué esquina se encuentra.

Como aclaración previa para entender el ejemplo, cabe matizar que el proceso de identificación resulta costoso en cuanto a memoria y CPU. Para optimizar dicho proceso, una técnica a emplear es la de manipular imágenes en escala de grises y que además sean de menor tamaño que la original. Con ello, el tiempo de cómputo se reduce considerablemente, permitiendo una interacción más cómoda.

Llegados a este punto, el lector entenderá gran parte del código, de modo que nos centraremos en aquellas partes que resultan nuevas. Así, en el bucle que recoge los frames de la captura, el clasificador es cargado sólo la primera vez [32-34]. A continuación, para cada frame capturado, se realizará una transformación a escala de grises [36-37], se reducirá la imagen a otra más pequeña [39-42] y finalmente se ecualizará el histograma [43]. Este último paso de ecualizar el histograma, sirve para tratar de resaltarán detalles que pudieran haberse perdido en los procesos de transformación a escala de grises y reducción de tamaño. Tras finalizar el proceso de identificación de rostros invocando la primitiva *cvHaarDetectObjects* [45-47], si no se ha encontrado ninguno se muestra por pantalla el mensaje *"Buscando objeto..."* empleando para ello la primitiva de *cvPutText* [39], la cual recibe la imagen donde se pondrá el texto, el texto propiamente dicho, la fuente que se empleará y que fue inicializada en la línea [25], y el color con el que se pintará. Si se han identificado rostros en la escena, mostrará el mensaje *"Objetivo encontrado!"* [53]. Si el rostro está en el cuarto derecho o izquierdo de la pantalla, mostrará el mensaje oportuno [59-64], además de identificarlo mediante un recuadro [66].

Listado 31.10: Interacción mediante reconocimiento

```
1  #include "highgui.h"
2  #include "cv.h"
3  #include "stdio.h"
4
5  int main( int argc, char** argv ) {
6    CvCapture* capture;
7    if( argc != 2 || !(capture = cvCreateFileCapture(argv[1])) )
8      capture = cvCreateCameraCapture(0);
9
10   static CvScalar colors[] = { {{0,0,255}},   {{0,128,255}},
11                   {{0,255,255}}, {{0,255,0}},
12                   {{255,128,0}}, {{255,255,0}},
13                   {{255,0,0}},   {{255,0,255}} };
14
15   cvNamedWindow( "Window", CV_WINDOW_AUTOSIZE );
16
17   IplImage *frame, *gray, *small_img;
18   double scale = 2;
19
20   CvMemStorage* storage = cvCreateMemStorage(0);
21   cvClearMemStorage( storage );
22   CvHaarClassifierCascade* classifier = NULL;
23
24   CvFont font;
25   cvInitFont(&font, CV_FONT_HERSHEY_SIMPLEX,
26          0.75, 0.75, 0, 2, CV_AA);
27
28   while(1) {
29     frame = cvQueryFrame(capture);
30     if( !frame ) break;
31
32     if (classifier == NULL)
33       classifier = cvLoadHaarClassifierCascade
34     ("./media/haarcascade_frontalface_alt.xml", cvGetSize(frame));
35
36     gray = cvCreateImage( cvGetSize(frame), 8, 1 );
37     cvCvtColor( frame, gray, CV_BGR2GRAY );
38
39     small_img = cvCreateImage( cvSize( cvRound (gray->width/scale),
40                         cvRound (gray->height/scale)),
41                   8, 1 );
42     cvResize( gray, small_img, CV_INTER_LINEAR );
43     cvEqualizeHist( small_img, small_img );
44
45     CvSeq* faces = cvHaarDetectObjects
46       (small_img, classifier, storage, 1.1, 4, 0,
```

```
47          cvSize( 40, 50 ), cvSize( 40, 50 ));
48
49      if (!faces->total)
50        cvPutText(frame, "Buscando objetivo...",
51          cvPoint(10, 50), &font, cvScalar(255, 255, 255, 0));
52      else
53        cvPutText(frame, "Objetivo encontrado!",
54          cvPoint(10, 50), &font, cvScalar(0, 0, 255, 0));
55
56      int i; CvRect* r;
57      for( i = 0; i < (faces ? faces->total : 0 ); i++ ){
58        r = (CvRect*)cvGetSeqElem(faces, i);
59        if (r->x < (small_img->width*0.25))
60      cvPutText(frame, "En la izquierda!",
61            cvPoint(10, 100), &font, cvScalar(0, 255, 255, 0));
62        if ((r->x+r->width) > (small_img->width*0.75))
63      cvPutText(frame, "En la derecha!",
64            cvPoint(10, 100), &font, cvScalar(0, 255, 255, 0));
65
66        cvRectangle(frame, cvPoint( r->x * scale, r->y * scale ),
67            cvPoint((r->x + r->width) * scale,
68                (r->y + r->height) * scale),
69            colors[i%8],3,10,0);
70      }
71
72      cvShowImage( "Window", frame );
73      cvReleaseImage( &gray );
74      cvReleaseImage( &small_img );
75
76      char c = cvWaitKey(5);
77      if(c == 27) break;
78    }
79
80    cvReleaseCapture(&capture);
81    cvDestroyWindow("Window");
82
83    return 0;
84  }
```

En la figura 31.6 puede verse la salida de la aplicación cuando identifica un rostro a la izquierda o a la derecha.

Figura 31.6: La aplicación muestra cuándo ha identificado un rostro y si este se encuentra a la izquierda o a la derecha.

31.7. Interacción mediante detección del color de los objetos

La detección de objetos es un proceso costoso en tiempo y cómputos. Además, si es necesaria una interacción mediante objetos de los cuales no disponemos de un clasificador ni de ejemplos suficientes con los que realizar su entrenamiento para la posterior fase de reconocimiento, existen otras alternativas. Una de ellas es identificar objetos que tengan un determinado color único en la escena.

Para ello, deberán realizarse diferentes transformaciones sobre la imagen, de modo que se filtre todo aquello que tenga un color diferente al del objeto que se desea detectar.

A modo de ejemplo, se construirá una sencilla aplicación en la que se irá pintando el trazo por el lugar por donde va pasando un objeto rastreado. El código es el que se lista a continuación:

Listado 31.11: Detección de objeto con determinado color

```
#include <cv.h>
#include <highgui.h>

int main(){
  CvCapture *capture = cvCaptureFromCAM(0);

  if(!capture) return -1;

  cvNamedWindow("Window", CV_WINDOW_FULLSCREEN);

  IplImage* frame = NULL;
  IplImage* imgScribble = NULL;
  while(1){
    frame = cvQueryFrame(capture);
    if(!frame) break;

    if(imgScribble == NULL)
    imgScribble = cvCreateImage(cvGetSize(frame), 8, 3);

    cvSmooth(frame, frame, CV_GAUSSIAN, 11, 11, 0, 0);

    IplImage* imgHSV = cvCreateImage(cvGetSize(frame), 8, 3);
    cvCvtColor(frame, imgHSV, CV_BGR2HSV);

    IplImage* imgThreshed = cvCreateImage(cvGetSize(frame), 8, 1);
    cvInRangeS(imgHSV, cvScalar(25, 100, 100, 0),
           cvScalar(40, 255, 255, 0), imgThreshed);

    CvMoments *moments = (CvMoments*)malloc(sizeof(CvMoments));
    cvMoments(imgThreshed, moments, 0);

    double moment10 = cvGetSpatialMoment(moments, 1, 0);
    double moment01 = cvGetSpatialMoment(moments, 0, 1);
    double area     = cvGetCentralMoment(moments, 0, 0);

    static int posX = 0; static int posY = 0;
    int lastX = posX; int lastY = posY;
    posX = moment10/area; posY = moment01/area;

    if(lastX>0 && lastY>0 && posX>0 && posY>0)
      cvLine(imgScribble,
         cvPoint(posX, posY), cvPoint(lastX, lastY),
         cvScalar(255, 255, 0, 0), 5, 1, 0);

    cvAdd(frame, imgScribble, frame, 0);
    cvShowImage("Window", frame);

```

```
48      char c = cvWaitKey(5);
49      if(c==27) break;
50
51      cvReleaseImage(&imgHSV);
52      cvReleaseImage(&imgThreshed);
53      free(moments);
54    }
55
56    cvReleaseCapture(&capture);
57    cvDestroyWindow("Window");
58
59    return 0;
60  }
```

En él puede verse cómo, una vez seleccionado el color del objeto a detectar, el procedimiento a llevar a cabo es el siguiente (la salida de su ejecución aparece en la figura 31.9):

1. Realizar un proceso de suavizado para eliminar el posible "ruido" en el color `21`.
2. Convertir la imagen RGB del frame capturado a una imagen HSV (Hue, Saturation, Value) (Matiz, Saturación, Valor) de modo que sea más fácil identificar el color a rastrear `23-24`. La salida de este filtro es la que se muestra en la parte izquierda de la figura 31.8.
3. Eliminar todos aquellos pixels cuyo matiz no corresponda con el del objeto a detectar. Esto devolverá una imagen en la que serán puestos a 0 todos los pixels que no estén dentro de nuestro criterio. En definitiva, la imagen sólo contendrá aquellos pixels cuyo color deseamos detectar `25-27` en color blanco y el resto todo en negro. La salida de este proceso es la que se muestra en la parte derecha de la figura 31.8.
4. Identificar el "momento" de los objetos en movimiento, es decir, su posición en coordenadas `29-34`.
5. Pintar una linea desde el anterior "momento" hasta el "momento" actual `29-43` en una capa (en el código es una imagen denominada *"imgScribble"*).
6. Unir la capa que contiene el trazo, con aquella que contiene la imagen capturada desde la cámara `45` y mostrarlo por pantalla `46`.

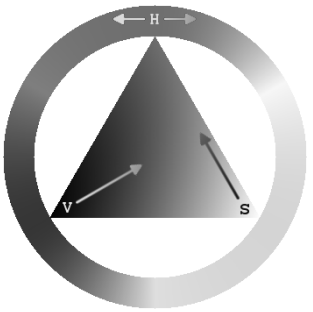


Figura 31.7: Triángulo HSV donde se representa el matiz (en la ruleta), la saturación y el valor (en el triángulo)

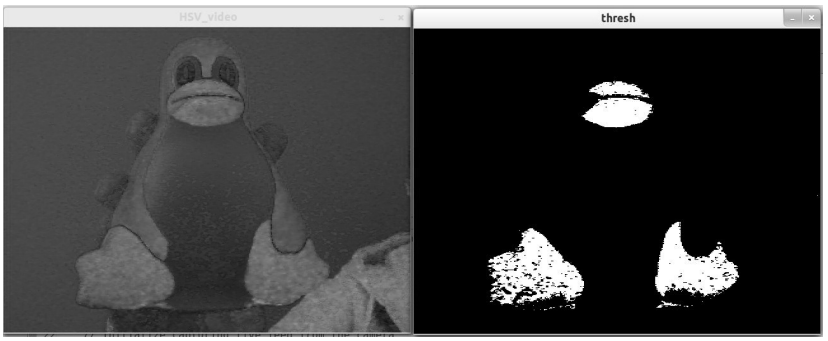


Figura 31.8: A la izquierda, se muestra la imagen HSV del Tux capturado con la cámara. A la derecha sólo aquello que coincida con el valor del color de las patas y el pico (amarillo en nuestro caso)

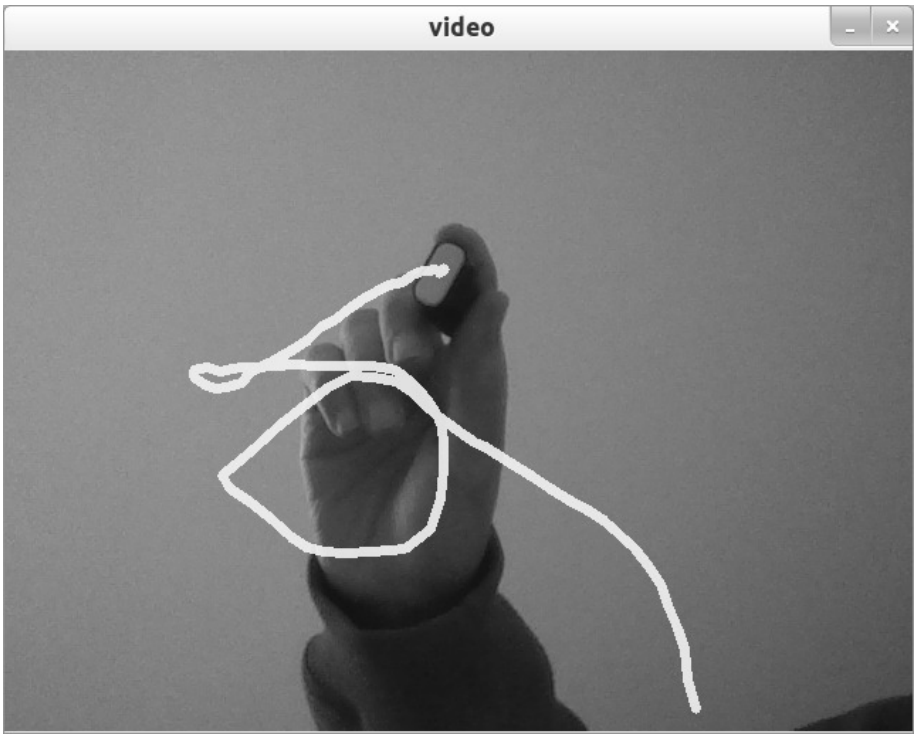


Figura 31.9: Salida de la ejecución del ejemplo de detección de objeto con determinado color. Puede verse cómo se superpone a la imagen el trazo realizado por el rastreo de un pequeño objeto amarillo fluorescente.

31.8. Identificación del movimiento

Para explicar la identificación del movimiento, tomaremos uno de los ejemplos que se proporcionan junto a la librería OpenCV. Se trata del ejemplo *motempl*, del cual se dispone de implementación en C y Python. En él se analiza tanto el proceso de identificación del movimiento, como el de la detección de la dirección y sentido del mismo.

Listado 31.12: Bucle principal del ejemplo *motempl.c*

```
1  for(;;)
2    {
3      IplImage* image = cvQueryFrame( capture );
4      if( !image )
5        break;
6
7      if( !motion )
8        {
9      motion = cvCreateImage( cvSize(image->width,image->height), 8,
           3 );
10     cvZero( motion );
11     motion->origin = image->origin;
12       }
13
14     update_mhi( image, motion, 30 );
15     cvShowImage( "Motion", motion );
16
17     if( cvWaitKey(10) >= 0 )
```

```
18        break;
19    }
```

Para realizar el análisis de dicho código debemos comenzar por el bucle principal. En él puede verse cómo se captura la imagen, y si aún no existe movimiento detectado, se crea una nueva imagen cuyo origen se hará coincidir con el del primer *frame* capturado [7-12]. A continuación, se actualiza el historial de movimiento (MHI (Motion History Image)) en la función *update_mhi* y se muestra la imagen por pantalla [15]. La función *update_mhi* recibe tres parámetros: la imagen actual procedente de la captura, la imagen resultante del movimiento y un umbral que establece a partir de cuándo la diferencia entre dos frames se considerará movimiento.

Listado 31.13: Identificación de silueta en *motempl.c*:

```
1  void  update_mhi(IplImage* img, IplImage* dst, int diff_threshold){
2    ...
3    // allocate images at the beginning or
4    // reallocate them if the frame size is changed
5    ...
6    cvCvtColor( img, buf[last], CV_BGR2GRAY ); // convert frame to
         grayscale
7
8    idx2 = (last + 1) % N; // index of (last - (N-1))th frame
9    last = idx2;
10
11   silh = buf[idx2];
12   cvAbsDiff( buf[idx1], buf[idx2], silh ); // get difference
         between frames
13
14   cvThreshold( silh, silh, diff_threshold, 1, CV_THRESH_BINARY );
         // and threshold
         it
15   cvUpdateMotionHistory( silh, mhi, timestamp, MHI_DURATION ); //
         update MHI
16
17   // convert MHI to blue 8u image
18   cvCvtScale( mhi, mask, 255./MHI_DURATION,
19           (MHI_DURATION - timestamp)*255./MHI_DURATION );
20   cvZero( dst );
21   cvMerge( mask, 0, 0, 0, dst );
22   ...
23 }
```

En el código de la función *update_mhi* puede verse cómo se inicializan algunas variables y se reserva la memoria para las imágenes. Tras esto lo interesante viene a continuación. En la línea [6] puede verse cómo la imagen de entrada es transformada a escala de grises y almacenada al final de una lista circular (en este caso de sólo cuatro elementos dado que N está declarado como *const int N = 4*). Tras actualizar los índices de la lista circular, se obtiene la diferencia entre los *frames* [12] mediante la llamada a la primitiva *cvAbsDiff*.

Ésta proporciona una imagen donde la salida es la diferencia de los pixels entre una imagen y la siguiente. Aquellos pixels con diferencia distinta de cero, son los que han cambiado y por tanto los que se han movido. Bastará con aplicar un filtrado *threshold* binario [14] al resultado de la operación anterior, de manera que todos aquellos pixels cuya diferencia sea mayor que un determinado umbral tomarán el valor 1 y en caso contrario el valor 0. En este punto se dispone de la "silueta" de aquella parte de la imagen que se ha movido (aquí la imagen llamada *silh*).

El siguiente paso es el de actualizar la imagen del movimiento con la silueta obtenida. Para ello se emplea la primitiva *cvUpdateMotionHistory* que recibe como parámetros:

- Máscara de la silueta que contiene los pixels que no están puestos a 0 donde ocurrió el movimiento.
- Historia de imágenes del movimiento, que se actualiza por la función y debe ser de un único canal y 32 bits.
- Hora actual.
- Duración máxima para el rastreo del movimiento.

Una vez obtenida la silueta en movimiento (el MHI), se crea una imagen donde se mostrará la misma en color azul (17-21).

El siguiente reto del código incluido en *motempl.c* es el de determinar cuál es la dirección y sentido del movimiento realizado. El fragmento de código encargado de esta funcionalidad puede verse en el siguiente listado.

Listado 31.14: Calculando dirección del movimiento en *motempl.c*

```
1   // calculate motion gradient orientation and valid orientation
        mask
2   cvCalcMotionGradient( mhi, mask, orient, MAX_TIME_DELTA,
        MIN_TIME_DELTA, 3 );
3
4   if( !storage )
5     storage = cvCreateMemStorage(0);
6   else
7     cvClearMemStorage(storage);
8
9   // segment motion: get sequence of motion components
10  // segmask is marked motion components map. It is not used
        further
11  seq = cvSegmentMotion( mhi, segmask, storage, timestamp,
        MAX_TIME_DELTA );
12
13  // iterate through the motion components,
14  // One more iteration (i == -1) corresponds to the whole image
        (global motion)
15  for( i = -1; i < seq->total; i++ ) {
16
17    if( i < 0 ) { // case of the whole image
18      comp_rect = cvRect( 0, 0, size.width, size.height );
19      color = CV_RGB(255,255,255);
20      magnitude = 100;
21    }
22    else { // i-th motion component
23      comp_rect = ((CvConnectedComp*)cvGetSeqElem( seq, i ))->rect;
24      if( comp_rect.width + comp_rect.height < 100 ) // reject very
            small components
25    continue;
26      color = CV_RGB(255,0,0);
27      magnitude = 30;
28    }
29
30    // select component ROI
31    cvSetImageROI( silh, comp_rect );
32    cvSetImageROI( mhi, comp_rect );
33    cvSetImageROI( orient, comp_rect );
34    cvSetImageROI( mask, comp_rect );
35
36    // calculate orientation
37    angle = cvCalcGlobalOrientation( orient, mask, mhi, timestamp,
          MHI_DURATION);
38    angle = 360.0 - angle;  // adjust for images with top-left
          origin
39
40    count = cvNorm( silh, 0, CV_L1, 0 ); // calculate number of
          points within silhouette ROI
```

```
41
42      cvResetImageROI( mhi );
43      cvResetImageROI( orient );
44      cvResetImageROI( mask );
45      cvResetImageROI( silh );
46
47      // check for the case of little motion
48      if( count < comp_rect.width*comp_rect.height * 0.05 )
49        continue;
50
51      // draw a clock with arrow indicating the direction
52      center = cvPoint( (comp_rect.x + comp_rect.width/2),
53                (comp_rect.y + comp_rect.height/2) );
54
55      cvCircle( dst, center, cvRound(magnitude*1.2), color, 3, CV_AA,
            0 );
56      cvLine( dst, center, cvPoint( cvRound( center.x + magnitude*cos
            (angle*CV_PI/180)),
57                    cvRound( center.y - magnitude*sin(angle*CV_PI
                          /180))), color, 3, CV_AA, 0 );
58    }
```

En él puede apreciarse cómo para obtener la dirección comienza calculando el gradiente de la orientación del MHI con *cvCalcMotionOrientation* [2]. A continuación trocea todo el movimiento en partes separadas, a fin de identificar los diferentes componentes en movimientos con la función *cvSegmentMotion* [11]. Así, iterando sobre cada uno de los componentes del movimiento identificados, puede calcularse la dirección concreta del movimiento de cada componente con la función *cvCalcGlobalOrientation* [37] empleando la máscara extraída para cada componente concreto. Para pasar a la función sólo aquellas partes de la imagen que intervienen en cada componente, en el código se extraen las regiones de interés (ROI) [30–34].

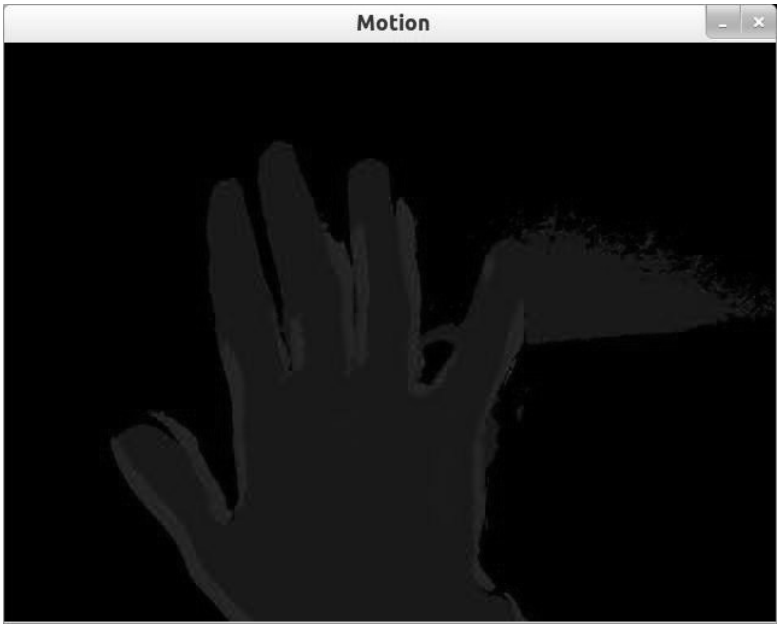


Figura 31.10: Detección del movimiento capturado. En azul aquellos pixels donde se ha localizado movimiento. En negro el fondo estático.

Finalmente marcará con un reloj la dirección de cada componente del movimiento. Para ello construye un círculo en torno a cada componente y dibuja una línea en su interior para indicar el ángulo del movimiento [51–57]. En la figura 31.10 puede encontrarse una salida con la imagen en azul del MHI y el resto de elementos estáticos en negro.

31.9. Comentarios finales sobre Visión por Computador

A lo largo de las secciones anteriores se han introducido algunos de los conceptos básicos de las técnicas de visión por computador y de cómo estas pueden ayudarnos a construir interfaces naturales que nos permitan interactuar con nuestras aplicaciones y juegos.

Para ello se ha abordado la librería OpenCV, centrándonos desde el comienzo en el procesamiento de vídeo y en la captura del mismo procedente de una WebCam. Así, tras hacer una introducción al filtrado y procesamiento de los frames que componen el vídeo, se han mostrado algunos conceptos básicos sobre interacción a partir de identificación de objetos como rostros, ojos, etc., reconocimiento de objetos con un determinado color y detección del movimiento. Si desea profundizar más en algunos temas relacionados con la visión por computador, animamos a la lectura de [15].

Con esto esperamos que el alumno haya adquirido una panorámica general sobre las capacidades que la visión por computador puede ofrecer al paradigma de la interacción natural aplicada al mundo de los videojuegos.

31.10. Caso de estudio: Wiimote

En el año 2006, Nintendo lanzó su consola Wii con un dispositivo hardware que permitía realizar interacción natural con el usuario: el Wiimote o mando a distancia de Wii.

El Wiimote realiza la detección del movimiento del usuario y transfiere esta información al computador o la consola. La correcta interpretación de los datos que se reciben desde este dispositivo, permitirá que las aplicaciones puedan implementar interfaces de usuario naturales, en las que un elemento externo al juego o a la aplicación, pueda interaccionar con el mundo virtual.

Entender el funcinamiento interno del mando y conocer el funcionamiento de librerías que nos permitan manipular el mando será el objetivo de las siguientes secciones.

31.11. Descripción del mando de Wii

Aunque a simple vista el Wiimote tenga un aspecto parecido al del mando a distancia de la televisión, éste cuenta con multitud de elementos sensores, actuadores y de comunicación que permiten detectar e identificar los movimientos gestuales de los usuarios. Enumerándolos, los elementos de los que consta el WiiMote son:

1. Elementos sensores
    - Tres acelerómetros para medir la fuerza ejercida en cada uno de los ejes.

- Una cámara infrarroja (IR) de cuatro puntos sencibles situada en el frontal superior para determinar el lugar al que el Wiimote está apuntando.
- Siete botones pulsables distribuidos a lo largo del mando (“A”, “1”, “2”, “+”, “-”, “HOME” y “POWER”).
- Cuatro botones de dirección en cruz (arriba, abajo, izquierda, derecha) situados en la zona superior.

2. Elementos actuadores

   - Cuatro luces LED (Light Emitter Diode) localizados en la zona inferior.
   - Un pequeño altavoz para emitir sonidos al usuario en base a respuestas de la aplicación, y que se encuentra en la zona central del mando.
   - Un motor de vibracion para enviar “zumbidos” al usuario en función de la interacción de este.

3. Interfaces de comunicación

   - Un puerto de expansion para permitir conectar otros dispositivos (como el Nunchuk) en el frontal inferior.
   - Una interfaz de conectividad Bluetooth, gracias a la cuál es posible emparejarlo con cualquier computador e implementar aplicaciones que interaccionen con él.

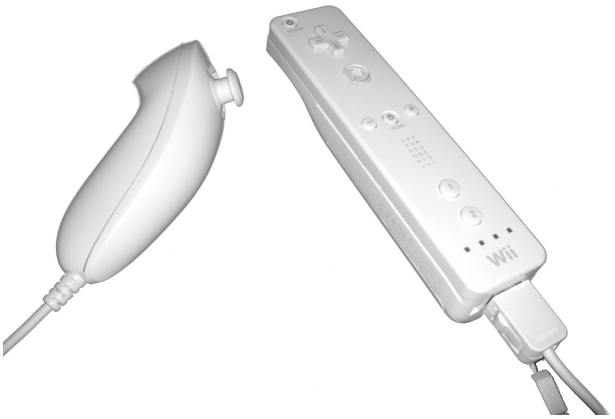


Figura 31.11: Una fotograría de un WiiMote (a la derecha) y el *Nunchunk* (a la izquierda).

Adicionalmente, el Wiimote viene acompañado de una barra de emisión de luz infrarroja (IR). Esta barra se coloca en la parte superior o inferior de la pantalla con la que se desee interactuar. Tiene 20 cm de longitud y cuenta con diez LED infrarrojos divididos en dos grupos, cinco en cada extremo de la barra. En cada grupo de cinco, los que se encuentran más al exterior están apuntando ligeramente hacia afuera, mientras que los más interiores apuntan ligeramente al centro. Los cuatro restantes apuntan hacia adelante. Esto permite conformar dos fuentes de luz IR bien diferenciadas.

Como se ha mencionado, el Wiimote cuenta con un sensor de IR localizado en el frontal superior del mando. Así, este sensor es capaz de localizar las dos fuentes de luz IR procedentes de los extremos de la barra y saber a qué posición concreta de la pantalla está apuntando el usuario hasta una distancia de unos cinco metros respecto de la barra.

A través del puerto de expansión situado en la parte inferior del mando, pueden conectarse una serie de dispositivos adicionales para permitir incrementar las posibilidades de interacción que proporciona. Entre estos dispositivos caben destacar el *Nunchunk*, el cuál consiste en un *joystick* analógico con dos botones más (comúnmente utilizado para controlar el movimiento de los personajes), la *Wii Wheel*, que transforma el mando en un volante para aquellos juegos de simulación de vehículos, o el *Wii Zapper*, que convierte al Wiimote+Nunchuk en una especie de "pistola".

El objetivo de todos estos elementos es siempre el mismo: *"proporcionar una interacción lo más cómoda y natural posible al usuario a partir de sensores y actuadores que obtengan información acerca del movimiento que éste está realizando"*.

Interpretar esta información de forma correcta y crear los manejadores de eventos apropiados para programar la aplicación en función de las necesidades de interacción, corre a cargo de los programadores de dichas apliciaciones. Para podeer capturar desde nuestras aplicaciones la información procedente de un dispositivo Wiimote, este dispone del puerto Bluetooth. Así, podrán programarse sobre la pila del protocolo Bluetooth las correspondientes librerías para comunicarse con el mando. En el siguiente apartado se abordará cómo trabajar con éste tipo de librerías.

31.12. Librerías para la manipulación del Wiimote

Lo cierto es que dependiendo del sistema operativo en el que nos encontremos deberemos optar por unas librerías u otras. Sin embargo, existen similitudes muy estrechas entre todas ellas, por lo que no resulta nada complicado migrar de una a otra.

Para los siguientes ejemplos se hará uso de la librería *libwiimote* disponible bajo licencia GPL en `http://libwiimote.sourceforge.net/`. Esta se encuentra escrita en C y proporciona un API para sistemas GNU/Linux.

Libwiimote aporta mecanismos para leer los datos de los acelerómetros, del sensor de IR de los estados de pulsación de los botones, del estado de la batería y del *nunchuk* (su botón, acelerómetros y estados del joystick). Asimismo, permite enviar señales o eventos hacia el mando para activar y desactivar los actuadores, de modo que se puedan establecer los LEDs a encendido/apagado, lanzar "zumbidos" o reproducir sonidos en el pequeño altavoz.

En las siguientes subsecciones se mostrará como manipular dicha librería.

Estructura *wiimote_t*

Libwiimote define una estructura directamente accesible desde nuestra aplicación en la que se almacena el estado de los distintos elementos del mando. En nuestras aplicaciones necesitaremos una estructura de este tipo por cada uno de los Wiimotes que se encuentren conectados.

Esta estructura es la *wiimote_t* que se encuentra definida en el fichero *wiimote.h*. Todos los elementos del mando se encuentran directamente accesibles a través de los campos de esta estructura. Su definición es la que se muestra en el siguiente listado:

Listado 31.15: Estructura *wiimote_t*

```
1 typedef struct {
2     wiimote_mode_t mode; /* Current report mode of wiimote. */
3     wiimote_keys_t keys;    /* Current key state. */
4     wiimote_point3_t axis;  /* Current accelerometer data. */
```

```
5      wiimote_ir_t ir1;       /* First detected IR source. */
6      wiimote_ir_t ir2;       /* Second detected IR source. */
7      wiimote_ir_t ir3;       /* Third detected IR source. */
8      wiimote_ir_t ir4;       /* Fourth detected IR source. */
9
10     wiimote_cal_t cal;      /* Wiimote calibration data */
11     wiimote_ext_port_t ext; /* Current extension port state. */
12
13     wiimote_link_t link;    /* Current link state. */
14     wiimote_led_t led;      /* Current state of the leds. */
15     uint8_t rumble;         /* Current state of rumble. */
16     uint8_t speaker;        /* ... */
17     uint8_t battery;        /* Current battery status. */
18
19     wiimote_float3_t tilt;  /* The tilt of the wiimote in degrees*/
20     wiimote_float3_t force; /* The force in g on each axis. */
21
22     struct {
23         wiimote_mode_t mode;
24         wiimote_keys_t keys;
25         wiimote_led_t led;
26         uint8_t rumble;
27     } old;
28 } __attribute__((packed)) wiimote_t;
```

Puede verse cómo quedan representados los botones, los acelerómetros, ángulos de inclinación, fuerzas de gravedad, el altavoz, la batería, etc.

Bucle principal de captura de datos y comunicación con el mando

Toda aplicación que manipule el Wiimote deberá contar con un bucle de captura/envío de datos desde y hacia el mando. Este tiene el aspecto que se muestra en el listado analizado a continuación.

En él código se incluye la cabecera de la librería y en un momento dado se inicializa la estructura *wiimote_t* (5) y se conecta con el mando mediante la primitiva *wiimote_connect* (6), a la que se le pasa una referencia a la anterior estructura y la dirección MAC del mando. Si desconocemos la dirección del mando, en GNU/Linux podemos emplear el comando *hcitool* mientras pulsamos simultaneamente los botones 1 y 2 del mando.

```
# hcitool scan
```

En caso de error, mediante la primitiva *wiimote_get_error*, podrá obtenerse información de la causa. Si todo ha ido bien, mientras la conexión siga abierta (*wiimote_is_open*) (13), la aplicación permanecerá dentro de un bucle en el que se resincronizará con el mando mediante *wiimote_update* (14), capturando información del mando y provocando la activación de los actuadores si procede. Como se muestra en el código, el usuario puede desconectarse del mando mediante la primitiva *wiimote_disconect* (16), que en el caso del ejemplo se ejecuta cuando se pulsa el botón *home*.

Listado 31.16: Bucle principal de captura/envío de datos

```
1 #include "wiimote_api.h"
2 ...
3
4 wiimote_t wiimote = WIIMOTE_INIT;
5 if (wiimote_connect(&wiimote, "XX:XX:XX:XX:XX:XX")<0){
6   fprintf(stderr, "Error al conectar con el mando: %s\n",
        wiimote_get_error());
```

```
  exit(1);
}

//Establecer el modo de captura
while (wiimote_is_open(&wiimote)) {
  wiimote_update(&wiimote);// sincronizar con el mando
  if (wiimote.keys.home) {   // alguna comprobación para salir
    wiimote_disconnect(&wiimote);
  }

  //Realizar captura de datos según el modo
  //Realizar envío de datos
}
```

Captura de pulsación de botones

Para detectar si el usuario está pulsando alguno de los botones del mando o del Nunchuk (incluido el joystick), podemos hacerlo comprobando directamente sus valores a través de los campos que se encuentran en *wiimote.keys* y *wiimote.ext.nunchuk* de la variable de tipo *wiimote_t*, tal y como se muestra en el ejemplo.

Listado 31.17: Detección de botores y joystick

```
#include <stdio.h>
#include <stdlib.h>
#include "wiimote_api.h"

int main(int argc, char **argv) {
    if (argc!=2){
        printf("Introduzca la mac del mando");
        exit(0);
    }else{
        printf("Pulse 1+2 para sincronizar");
    }

    wiimote_t wiimote = WIIMOTE_INIT;
    wiimote_connect(&wiimote, argv[1]);
    while (wiimote_is_open(&wiimote)) {
        wiimote_update(&wiimote);
        if (wiimote.keys.home) {
            wiimote_disconnect(&wiimote);
        }

        printf("WIIMOTE: KEYS  %04x 1=%d 2=%d A=%d B=%d <=%d >=%d
            ^=%d v=%d +=%d -=%d home=%d ",
               wiimote.keys.bits,
               wiimote.keys.one,  wiimote.keys.two,
               wiimote.keys.a,    wiimote.keys.b,
               wiimote.keys.left, wiimote.keys.right,
               wiimote.keys.up,   wiimote.keys.down,
               wiimote.keys.plus, wiimote.keys.minus,
               wiimote.keys.home);

        printf("NUNCHUK: JOY1 joyx=%03d joyy=%03d keys.z=%d keys.c
            =%d\n",
               wiimote.ext.nunchuk.joyx,
               wiimote.ext.nunchuk.joyy,
               wiimote.ext.nunchuk.keys.z,
               wiimote.ext.nunchuk.keys.c);
    }

    return 0;
}
```

Captura de datos de los acelerómetros

Para cada uno de los ejes, el Wiimote proporciona valores recogidos por un acelerómetro. De este modo, si ponemos el Wiimote en horizontal sobre una mesa los valores recogidos por los acelerómetros tenderán a cero.

Es preciso tener en cuenta que los acelerómetros, como su nombre indica, miden la "aceleración", es decir, la variación de la velocidad con respecto del tiempo. No miden la velocidad ni la posición. Valores altos de uno de los acelerómetros indicará que el mando está variando muy deprisa su velocidad en el eje correspondiente. Que el acelerómetro proporcione un valor cero, indicará que la velocidad permaneciera constante, no que el mando esté detenido.

Además de la aceleración, resulta interesante medir los ángulos de inclinación del mando, y en consecuencia, la dirección en la que se ha realizado un movimiento. En el Wiimote, estos ángulos se conocen como cabeceo (*Pitch*) arriba y abajo en el eje x, balanceo o alabeo (*Roll*) en el eje y, y guiñada o viraje a izquierda y derecha (*Yaw*) en el eje z (ver figura 31.12).

Los movimientos de cabeceo (*Pitch*) y balanceo (*Roll*) pueden medirse directamente por los acelerómetros, pero la guiñada (*Yaw*) precisa de la barra de LEDs IR para permitir estimar el ángulo de dicho movimiento.

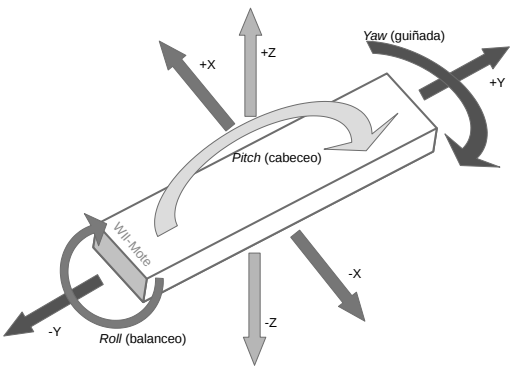


Figura 31.12: Ejes y ángulos de balanceo, viraje y cabeceo en el Wiimote.

Afortunadamente, el cálculo para la obtención de datos como la aceleración, las fuerzas G de la gravedad, y los ángulos de orientación, son funciones que habitualmente implementan las APIs que controlan el Wiimote. Así, para capturar los datos procedentes del acelerómetro, el modelo de trabajo es identico al mostrado con anterioridad. La salvedad es que antes de comenzar la captura de los datos, deberá activarse el flag *wiimote.mode.acc*. Una vez hecho esto, la librería pone a nuestra disposición tres grupos de valores x, y y z accesibles a través de la estructura *wiimote_t*:

- Ejes (*axis*): para proporcionar la aceleración en cada uno de los ejes, es decir, su vector aceleración.
- Ángulos (*tilt*): para indicar el ángulo de inclinación del acelerómetro para el balanceo, el cabeceo y el viraje.
- Fuerzas (*force*): para representar al vector fuerza de la gravedad en sus componentes (x, y, z).

El siguiente ejemplo muestra la recogida de estos datos.

Listado 31.18: Recoger datos del acelerómetro

```
#include <stdio.h>
#include <stdlib.h>
#include "wiimote_api.h"

int main(int argc, char **argv) {
    if (argc!=2){
        printf("Introduzca la mac del mando");
        exit(0);
    }else{
        printf("Pulse 1+2 para sincronizar");
    }

    wiimote_t wiimote = WIIMOTE_INIT;
    wiimote_connect(&wiimote, argv[1]);
    wiimote.mode.acc = 1; // habilitar acelerometro

    while (wiimote_is_open(&wiimote)) {
        wiimote_update(&wiimote);
        if (wiimote.keys.home) {
            wiimote_disconnect(&wiimote);
        }

        printf("AXIS x=%03d y=%03d z=%03d "
                "FORCE x=%.3f y=%.3f z=%.3f "
                "TILT x=%.3f y=%.3f z=%.3f\n",
                wiimote.axis.x, wiimote.axis.y, wiimote.axis.z,
                wiimote.force.x,wiimote.force.y,wiimote.force.z,
                wiimote.tilt.x, wiimote.tilt.y, wiimote.tilt.z);
    }

    return 0;
}
```

Envio de eventos al mando

Para enviar eventos al mando, tales como activar y desactivar los LEDs, provocar "zumbidos" o reproducir sonidos en el altavóz, bastará con activar el correspondiente flag de la estructura *wiimote_t*, lo que accionará el actuador correspondiente en el Wiimote.

En el siguiente código se muestra cómo provocar determinados eventos en el mando al pulsar el correspondiente botón del mismo.

Listado 31.19: Enviando acciones al mando

```
#include <stdio.h>
#include <stdlib.h>
#include "wiimote_api.h"

int main(int argc, char **argv) {
    if (argc!=2){
        printf("Introduzca la mac del mando");
        exit(0);
    }else{
        printf("Pulse 1+2 para sincronizar");
    }

    wiimote_t wiimote = WIIMOTE_INIT;
    wiimote_connect(&wiimote, argv[1]);

```

```
16      while (wiimote_is_open(&wiimote)) {
17          wiimote_update(&wiimote);
18          if (wiimote.keys.home) {
19              wiimote_disconnect(&wiimote);
20          }
21
22          if (wiimote.keys.up)wiimote.led.one = 1;
23          else wiimote.led.one = 0;
24
25          if (wiimote.keys.down) wiimote.led.two = 1;
26          else wiimote.led.two = 0;
27
28          if (wiimote.keys.left) wiimote.led.three = 1;
29          else wiimote.led.three = 0;
30
31          if (wiimote.keys.right) wiimote.led.four = 1;
32          else wiimote.led.four = 0;
33
34          if (wiimote.keys.a) wiimote.rumble = 1;
35          else wiimote.rumble = 0;
36      }
37
38      return 0;
39  }
```

Datos a interpretar del Wiimote según la interacción

Con lo que se ha expuesto hasta el momento, es fácil ver cómo programar aplicaciones que tengan en cuenta los datos procedentes del mando de Wii. Veamos algunas líneas guía:

- Para empelar el mando como puntero sobre la pantalla de modo que se puedan acceder a los diferentes menus de una aplicación o apuntar a un objetivo concreto en la pantalla, solo tendrá que conocerse la posición a la que se señala.
- Si se desea emplear el mando a modo de "volante" para una aplicación de simulación automovilística, bastará procesar la variación de la fuerza sobre el eje y y quizá implementar algunas funciones mediante los botones para frenar, acelerar, etc.

Listado 31.20: Usando el mando como volante

```
1       while (wiimote_is_open(&wiimote)) {
2           wiimote_update(&wiimote);
3           if (wiimote.keys.home) {
4               wiimote_disconnect(&wiimote);
5           }
6
7           if (wiimote.force.y>0)
8               printf ("< %.3f grados ", wiimote.force.y);
9           else if (wiimote.force.y<0)
10              printf ("> %.3f grados ", wiimote.force.y);
11          else
12              printf ("= %.3f grados ", wiimote.force.y);
13
14          if (wiimote.keys.one){
15              printf ("frenando");
16              wiimote.rumble = 1;
17          }else{
18              wiimote.rumble = 0;
19          }
20          if (wiimote.keys.two) printf ("acelerando");
21          if (wiimote.keys.a)   printf ("nitro");
```

```
22
23          printf ("\n");
24      }
```

- Suponga que desea implementar un juego donde se deba lanzar una caña de pescar o simular el golpe de un palo de golf. Deberá tomar e interpretar del modo adecuado el "viraje" de la mano. Así, para el caso de la caña de pescar se monitorizará la fuerza de la gravedad ejercida en el eje z para saber si está hacia adelante y hacia atrás, siempre y cuando la inclinación en el eje y sea negativo (la caña debe estar arriba). Además, deberá calcularse la fuerza a partir de las componentes del vector gravedad.

Listado 31.21: Usando el mando como caña de pescar

```
1       while (wiimote_is_open(&wiimote)) {
2           wiimote_update(&wiimote);
3           if (wiimote.keys.home) {
4               wiimote_disconnect(&wiimote);
5           }
6
7           if (wiimote.tilt.y>0){ //Balanceo positivo ==>caña
                arriba
8               printf ("ca~na abajo %.3f\n", wiimote.tilt.y);
9           }else{
10              if (wiimote.force.z<0)
11                  printf ("atras %.3f\n", wiimote.force.z);
12              else
13                  printf ("adelante %.3f\n", wiimote.force.z);
14          }
15      }
```

- Si lo que se desea implementar es la simulación de un gesto para "golpear" algo (quizá a un adversario en un combate de boxeo), será preciso analizar no sólo la fuerza, sino también la información para posicionar el puño.
- Emular objetos como espadas, baras, etc. implica que deberá procesarse tanto la fuerza como el valor de la inclinación en cada uno de los ejes para poder posicionar perfectamente el objeto en el mundo virtual.

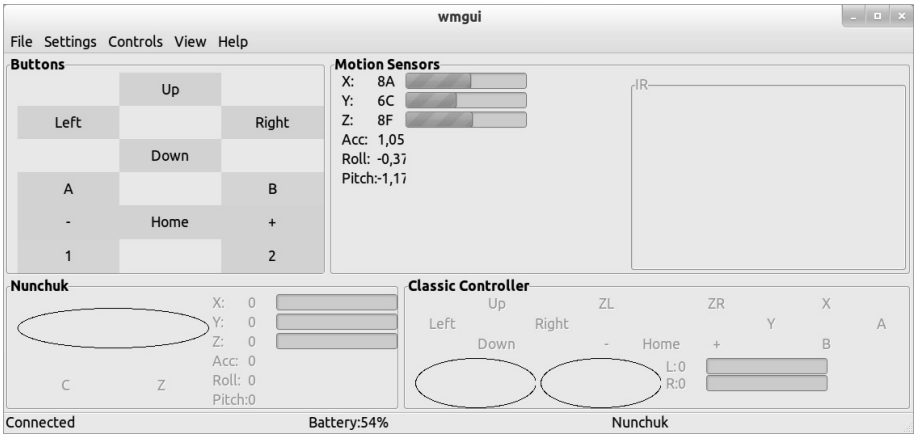


Figura 31.13: Captura de la aplicación WmGui.

Como puede verse las posibilidades son múltiples, solo es cuestión de identificar cuáles son los datos procedentes del mando que son imprescindibles para nuestra aplicación, posibilitándo de este modo la construcción de interfaces de usuario naturales, donde un elemento del mundo real externo a la aplicación, puede interaccionar con el mundo virtual dentro del computador.

Si en ocasiones no resulta evidente el conocer qué elementos deben procesarse y con qué valores, existen aplicaciones que pueden ayudar a testear cuáles son las variables que intervienen en un movimiento. En la figura 31.13 puede verse el aspecto de una de ellas: WmGui. Se trata de una aplicación para GNU/Linux donde podemos monitorizar parámetros como los botones pulsados, la variación sobre los ejes x, y, z, la acceleración como composición del vector gravedad, la rotación, el balanceo, etc.

31.13. Caso de estudio: Kinect

Siguiendo en la linea de dispositivos que permiten al usuario realizar una interacción natural con la consola o el computador, se encuentra Kinect. Se trata de un dispositivo desarrollado por Microsoft para su videoconsola Xbox 360, el cual permite a los programadores implementar aplicaciones capaces de reconocer gestos y emplear comandos mediante la voz.

El dispositivo Kinect está diseñado para poder localizarlo encima o debajo del monitor o del televisor con el que se interactuará. Como puede verse en la imagen 31.14, Kinect tiene una composición horizontal en la que se albergan:

- Una cámara RGB que proporciona una resolución VGA (Video Graphics Adapter) de 8 bits (640x480).
- Un sensor de profuncidad monocromo de resolución VGA de 11 bist que proporcina $2^{11} = 2048$ niveles de sensibilidad.
- Un array de cuatro micrófonos.

Figura 31.14: El dispositivo Kinect para XBox 360

El sensor de profuncidad consiste en una convinación de un laser infrarrojo y un sensor monocromo CMOS (Complementary Metal-Oxide-Semiconductor) de infrarrojos. Esto le permite capturar video 3D bajo cuálquier tipo de luz ambiental. El laser infrarrojo se proyecta sobre cada uno de los objetos que se encuentre a su alcance. El sensor recibe el infrarrojo reflejado por los objetos. Analizando las longitudes de onda de la señal reflejada, el dispositivo calcula la distancia a la que se encuentran los objetos.

Kinect cuenta además con un motor en su base, el cual le permite modificar su inclinación. Además dispone de una conexión USB (Universal Serial Bus), gracias a la cuál es posible comunicarnos con él e implementar aplicaciones que manipulen los datos que proporciona.

Hasta ahora Kinect no había tenido muchos rivales, sin embargo, cabe mencionar un par de competidores interesantes: ASUS Xtion Motion y PrimeSense Carmine. El primero ha sido desarrollado por ASUS y PrimeSense; el segundo, íntegramente por PrimeSense, la empresa que diseñó el sensor de profundidad de Kinect.

Desde el punto de vista de la programación, Kinect cuenta con un API desarrollada por Microsoft capaz de realizar detección y seguimiento de objetos –como cuerpo, cabeza, manos–, identificación de gestos, etc. Además, desde su aparición, tanto la liberación por parte de PrimeSense del código de los drivers del sensor de profuncidad, como la labor de ingeniería inversa desarrollada por algunas comunidades, han permitido que los desarrolladores puedan implementar APIs con las que poder manipular el dispositivo Kinect empleando alternativas libres.

En las siguientes secciones se abordarán algunas de las principales APIs y *Middlewares* de libre disposición más empleadas y robustas.

31.14. Comunidad OpenKinect

OpenKinect [1] es una comunidad interesada en manipular Kinect en sus aplicaciones. Desarrollan librerías de código abierto bajo licencia Apache2.0 y GPL2, para poder implementar aplicaciones que hagan uso de Kinect desde sistemas Windows, Linux y Mac. Su principal desarrollo es la librería *libfreenect*.

El código de *libfreenect* se encuentra disponible en la dirección `https://github.com/OpenKinect/libfreenect`, aunque se pueden encontrar los paquete para sistemas Ubuntu/Debian a través del correspondiente sistema de gestión de paquetes. Así, para instalar *libfreenect*, el paquete de desarrollo con sus cabeceras y las demos, bastará escribir desde consola:

```
# sudo add-apt-repository ppa:floe/libtisch
# sudo apt-get update
# sudo apt-get install freenect libfreenect libfreenect-dev
    libfreenect-demos
```

libfreenect requiere que el usuario que accede a Kinect pertenezca a los grupos *plugdev* y *video*. Deberá garantizarse que el usuario pertenece a estos grupos mediante los comandos:

```
# sudo adduser <USER> plugdev
# sudo adduser <USER> video
```

[1] `http://openkinect.org/wiki/Main_Page`

Tras esto, tendrá que cerrarse la sesión para volver a acceder de nuevo y hacer así efectivos los grupos a los que se ha añadido el usuario. Con ello, podrán ejecutarse las aplicaciones que manipulen Kinect, como las que se incluyen en el paquete de demostración para probar la captura realizada por las cámaras de Kinect:

```
# freenect-glview
```

En la figura 31.15 puede verse la salida de dicha aplicación. A la derecha se aprecia la imagen procedente de la cámara RGB, mientras que a la izquierda se muestran los niveles de profundidad representados por la escala de color.

Figura 31.15: Catura de pantalla de la ejecución de freenect-glview.

Con todo, lo que OpenKinect proporciona con *libfreenect* es un API para el acceso a los datos procedentes de Kinect y para el control del posicionamiento del motor. Sin embargo, procesar los datos leídos es responsabilidad de la aplicación, por lo que es común su empleo junto a librerías como OpenCV para la identificación del movimiento, reconocimiento de personas, etc.

Si lo que se desea es poder disponer de un API de más alto nivel, deberemos hacer uso de algún *framework*, como el de OpenNI que pasaremos a describir en la siguiente sección.

31.15. OpenNI

OpenNI (*Open Natural Interaction*) es una organización sin ánimo de lucro promovida por la industria para favorecer la compatibilidad e interoperabilidad de dispositivos, aplicaciones y *middlewares* que permitan la construcción de Interfaces Naturales.

La organización ha liberado un *framework* para el desarrollo de aplicaciones, el cual permite tanto la comunicación a bajo nivel con los dispositivos de video y audio de Kinect, como el uso de soluciones de alto nivel que emplean visión por computador para realizar seguimiento de objetos como manos, cabeza, cuerpo, etc.

El *framework* OpenNI se distribuyó bajo licencia GNU Lesser General Public License (LGPL (Lesser General Public License)) hasta la versión 1.5, pasando a distribuirse bajo Apache License desde la versión 2. Proporciona un conjunto de APIs para diferentes plataformas con *wrappers* para diversos lenguajes de programación entre los que están C, C++, Java y C#.

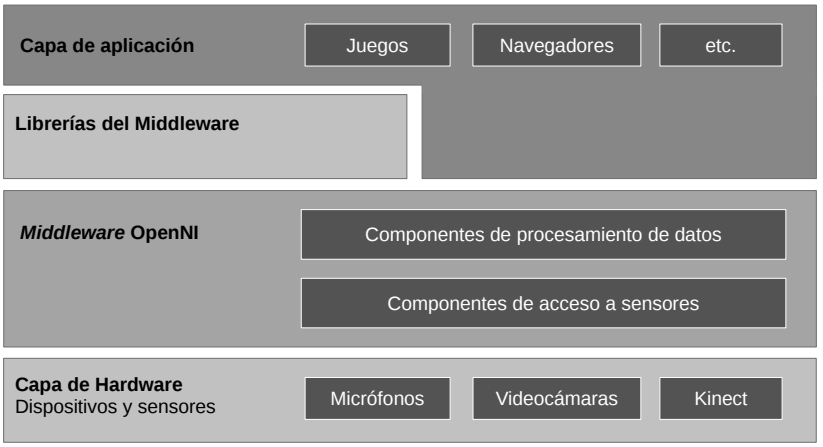


Figura 31.16: Arquitectura de OpenNI.

La arquitectura actual del *framework* de OpenNI (v2.0) es la que se muestra en la figura 31.16. En ella pueden verse cuatro capas bien diferenciadas:

- La capa en la que se encuentra el hardware o dispositivos sensores de audio y video.
- El *middleware* de OpenNI que permite:
  - Acceso a sensores de audio y vídeo, de modo que puedan recogerse los datos procedentes de los mismos.
  - Procesado de los datos procedentes de los sensores, de manera que puedan implementarse de forma cómoda determinados algoritmos de uso más habitual como identificación y rastreo de personas, manos, cabeza, etc.
- Las librerías del *middleware* ofrecidas por terceros para identificar y seguir partes del cuerpo, realizar reconocimiento de objetos, permitir reconstrucciones tridimensionales, y un largo etcétera. Estas hacen uso de *middleware* OpenNI.
- La capa de aplicación donde residen los programas que implementan interacción natural mediante el uso del *middleware* de OpenNI y las librerías proporcionadas por terceros.

El *middleware* OpenNI está formado por una serie de módulos o "componentes" que pueden ser registrados en el mismo. Estos permiten la lectura de los datos de los sensores (concretamente del sensor 3D, de la cámara RGB, de la cámara IR y de los micrófonos), y el procesamiento de los mismos para detección de cuerpos, manos, gestos, análisis de escena para identificar primer y segundo plano, separación de objetos, etc. Para esto último, destaca el *middleware* NiTE de Primesense que implementa multitud de detectores de gestos con una carga de CPU mínima, para plataformas Windows, Linux, Mac OS y Android.

31.15.1. Instalación

Para poder emplear OpenNI, deberá descargarse el software OpenNI, el *middleware* NiTE de PrimeSense y el driver del Sensor de la dirección `http://www.openni.org/openni-sdk/`, donde pueden encontrarse los paquetes preparados para Windows, Linux y Mac, así como el acceso a los repositorios *github* de los fuentes para ser compilados e instalados.

Sin embargo, si se desea desarrollar aplicaciones que empleen OpenNI en otras plataformas, una opción interesante y sencilla puede pasar por seguir el proyecto Simple-OpenNI[2], donde se pone a nuestra disposición una versión algo reducida del código de OpenNI con un *wrapper* para *Processing*, pero con funcionalidad sufiente para realizar una primera inmersión en el potencial de OpenNI.

Así, desde la página de Simple-OpenNI pueden descargarse ficheros zip con el código preparado para compilar e instalar en sistemas Linux de 64 bits, MacOSX y Windows de 32 y 64 bits. Una vez descargado el fichero zip para la plataforma correspondiente, bastará descomprimirlo y se dispondrá de directorios llamados *OpenNI-Bin-Dev-Linux-x64-v1.5.2.23*, *NITE-Bin-Dev-Linux-x64-v1.5.2.21* y *Sensor-Bin-Linux-x64-v5.1.0.41* si nos hemos descargado el código para una arquitectura de 64 bits.

En cualquiera de los casos, si se ha descargado el zip completo desde Simple-OpenNI o si se ha descargado desde los correspondientes repositorios *github*, para la instalación deberán compilarse e instalarse por separado cada uno de los anteriores paquetes, invocando al *./install* que se encuentra en cada uno de ellos:

1. OpenNI

```
# cd OpenNI-Bin-Dev-Linux-x64-v1.5.2.23
# sudo ./install
```

2. Nite

```
# cd NITE-Bin-Dev-Linux-x64-v1.5.2.21
# sudo ./install
```

3. Driver Kinect

```
# cd Sensor-Bin-Linux-x64-v5.1.0.41
# sudo ./install
```

31.15.2. Conceptos iniciales

Una vez instalado el software, antes de comenzar adentrándonos en las particularidades del *framework* OpenNI, deben definirse una serie de conceptos iniciales necesarios para comprender cómo programar aplicaciones que lo utilicen.

Módulos

Como se ha mencionado, el *framework* OpenNI proporciona una capa de abstracción tanto para dispositivos físicos como para otros componentes del *middleware*. Estos son conocidos como "módulos". OpenNI distingue dos grupos de módulos:

[2] `http://code.google.com/p/simple-openni/`

- Módulos sensores: encargados del sensor 3D, de la cámara RGB, de la cámara IR y de los micrófonos.
- Componentes del *middleware*: destinados al análisis y detección del cuerpo, puntos de la mano, detección de gestos, análisis de escena, etc.

Nodos de producción y generadores

Los nodos de producción son componentes que encapsulan una funcionalidad muy concreta. Estos toman unos datos de un tipo concreto en su entrada, para producir unos datos específicos de salida como resultado de un procesamiento concreto. Esta salida puede ser aprovechada por otros nodos de producción o por la aplicación final.

Existen nodos de producción tando de los sensores como del *middleware*. Aquellos nodos de producción que proporcionan datos que pueden ser manipulados directamente por nuestra aplicación son los conocidos como "generadores". Ejemplos de generadores son: el generador de imagen, que proporciona la imagen capturada por la cámara; el generador de profundidad, que proporciona el mapa de profundidad capturado por el sensor 3D; el generador de usuario, que proporciona datos sobre los usuarios identificados; el generador de puntos de la mano, que proporciona información acerca del muestreo de las manos; el generador de alertas de gestos, que proporciona datos a cerca de los gestos identificados en los usuarios, etc.

Cadenas de producción

Es la encadenación de varios nodos de producción de manera que la salida del procesamiento de uno es empleada como entrada para el procesamiento de otro, a fin de obtener el resultado final deseado.

31.15.3. Manipulando Kinect con OpenNI

Para conocer cuál es el modo habitual de proceder cuando se codifiquen aplicaciones que manipulen Kinect empleando OpenNI, veremos paso a paso un sencillo ejemplo donde se leen algunos datos del esqueleto de una persona. Comenzaremos analizando el código que se muestra en el listado.

Listado 31.22: Ej. sencillo para reconocimiento de la mano izquierda

```
1  int main(){
2    XnStatus nRetVal = XN_STATUS_OK;
3
4    //Inicialización del contexto
5    xn::Context context;
6    nRetVal = context.Init();
7
8    // Creación de los generadores necesarios
9    nRetVal = g_UserGenerator.Create(context);
10
11   // Establecer las opciones de los generadores
12   XnCallbackHandle h1, h2, h3;
13   g_UserGenerator.RegisterUserCallbacks(User_NewUser,
14                       User_LostUser, NULL, h1);
15   g_UserGenerator.GetPoseDetectionCap().RegisterToPoseCallbacks(
16                       Pose_Detected, NULL, NULL, h2);
17   g_UserGenerator.GetSkeletonCap().RegisterCalibrationCallbacks(
18                       Calibration_Start,
19                       Calibration_End, NULL, h3);
20
```

```
21    g_UserGenerator.GetSkeletonCap().SetSkeletonProfile(
22                        XN_SKEL_PROFILE_ALL);
23
24    // Hacer que los generadores comiencen su trabajo
25    nRetVal = context.StartGeneratingAll();
26
27    while (TRUE){
28      // Tomamos el siguiente frame
29      nRetVal = context.WaitAndUpdateAll();
30
31      // Para cada usuario detectado por el sistema
32      XnUserID aUsers[15];
33      XnUInt16 nUsers = 15;
34      g_UserGenerator.GetUsers(aUsers, nUsers);
35      for (int i = 0; i < nUsers; ++i){
36        // Recogemos los datos deseados del esqueleto
37        if (g_UserGenerator.GetSkeletonCap().IsTracking(aUsers[i])){
38          XnSkeletonJointPosition LeftHand;
39          g_UserGenerator.GetSkeletonCap().GetSkeletonJointPosition(
40         aUsers[i], XN_SKEL_LEFT_HAND, LeftHand);
41
42      // Procesamos los datos leídos
43      if (LeftHand.position.X<0) printf("RIGHT ");
44      else printf("LEFT ");
45
46      if (LeftHand.position.Y>0) printf("UP");
47      else printf("DOWN");
48
49      printf("\n");
50        }
51      }
52    }
53
54    context.Shutdown();
55
56    return 0;
57  }
```

En toda aplicación que manipule Kinect con el *framework* de OpenNI, se debe comenzar con la inicialización de un objeto "Contexto" [5-6]. Éste contiene todo el estado del procesamiento usando OpenNI. Cuando el Contexto no vaya a ser usado más, este deberá ser liberado [54].

Una vez que se tiene el objeto Contexto construido, se pueden crear los nodos de producción. Para ello será preciso invocar al constructor apropiado. Así se dispone de *xn::DepthGenerator::Create()* para generar datos de profundidad, *xn::-ImageGenerator::Create()* para generar imagen RGB, *xn::IRGenerator::Create()*, para generar datos del receptor de infrarrojos, *xn::AudioGenerator::Create()* para generar datos de audio procedente de los micrófonos, *xn::GestureGenerator::Create()* para generar datos de gestos, *xn::SceneAnalyzer::Create()* para generar datos de la escena, *xn::HandsGenerator::Create()* para generar datos de las manos, *xn::-UserGenerator::Create()* para generar datos del usuario, etc.

En el caso que nos ocupa, en el ejemplo se ha empleado un generador de usuario [9], el cual producirá información asociada a cada uno de los usuarios identificados en una escena, tales como el esqueleto o la posición de la cabeza, las manos, los pies, etc. Tras esto se deberán establecer las opciones que sean convenientes para procesar los datos del generador elegido. Volveremos a las opciones establecidas para el generador de usuario en breve, pero de momento continuaremos nuestra explicación obviando los detalles concretos de la inicialización de este.

Una vez establecidos las opciones de los generadores, deberá hacerse que estos comiencen a producir la información de la que son responsables. Esto se hará con la función *StartGeneratingAll()* del Contexto [25]. Tras esto se podrá entrar en el bucle principal de procesamiento de la información para recoger los datos

producidos por los generadores. En dicho bucle, lo primero que se hará es actualizar la información procedente de los generadores mediante alguna de las funciones de la familia *WaitAndUpdateAll()* del Contexto [29]. A continuación, se tendrá disponible la información proporcionada por los generadores, y que podrá recogerse con métodos *Get*, los cuales dependerán del generador o generadores que se estén manipulando.

En el ejemplo, lo que se hace es invocar al método *GetUsers* [34] que devuelve el Id para cada uno de los usuarios que ha reconocido el generador, así como el número de usuarios detectados en la escena. Con esta información, para cada uno de los usuarios identificados, se va recogiendo su esqueleto mediante el método *GetSkeletonCap()*, y a partir de este, la posición en la que se encuentra la mano izquierda con el método *GetSkeletonJointPosition()* [38] al que se le ha indicado qué parte del esqueleto se desea extraer (mediante el parámetro XN_SKEL_LEFT_HAND) y dónde se almacenará la información (en la estructura *LeftHand*). A partir de aquí, bastará con procesar la información de la estructura reportada [42-49].

Es preciso tener en cuenta que, cuando le pedimos datos del esqueleto al generador, es desde el punto de vista de la cámara. Es decir, la mano izquierda que "ve Kinect" se corresponderá con mano la derecha del usuario.

Como se ha mencionado con anterioridad, para poder disponer de los datos adecuados procedentes de los generadores, en ocasiones será preciso establecer ciertos parámetros de éstos y personalizar determinada funcionalidad. Ha llegado el momento de abordar de nuevo este fragmento del código. Así, en el ejemplo, en las líneas [11-22] puede verse cómo se han registrado determinados manejadores *XnCallbackHandle* que indican al generador de usuario qué acción debe realizar cuando se ha detectado un nuevo usuario o cuando se ha perdido uno (empleando para ello el método *RegisterUserCallbacks*), qué hacer cuando se ha registrado una determinada pose del usuario (mediante *RegisterToPoseCallbacks*), y las acciones a realizar al comenzar y terminar el proceso de calibración (usando *RegisterCalibrationCallbacks*).

Una implementación de estos manejadores o *callbacks* es la que se muestra en el siguiente código:

Listado 31.23: *Callback* para el generador de usuario

```
1  #define POSE_TO_USE "Psi"
2  xn::UserGenerator g_UserGenerator;
3
4  void XN_CALLBACK_TYPE
5  User_NewUser(xn::UserGenerator& generator,
6               XnUserID nId, void* pCookie){
7    printf("Nuevo usuario: %d\n", nId);
8    g_UserGenerator.GetPoseDetectionCap().
9      StartPoseDetection(POSE_TO_USE, nId);
10 }
11
12 void XN_CALLBACK_TYPE
13 User_LostUser(xn::UserGenerator& generator, XnUserID nId,
14              void* pCookie){}
15
16 void XN_CALLBACK_TYPE
17 Pose_Detected(xn::PoseDetectionCapability& pose,
18           const XnChar* strPose, XnUserID nId, void* pCookie){
19   printf("Pose %s para el usuario %d\n", strPose, nId);
20   g_UserGenerator.GetPoseDetectionCap().StopPoseDetection(nId);
21   g_UserGenerator.GetSkeletonCap().RequestCalibration(nId, TRUE);
22 }
23
24 void XN_CALLBACK_TYPE
25 Calibration_Start(xn::SkeletonCapability& capability, XnUserID nId,
26                   void* pCookie){
27   printf("Comenzando calibracion para el usuario %d\n", nId);
28 }
```

```
29
30 void XN_CALLBACK_TYPE
31 Calibration_End(xn::SkeletonCapability& capability, XnUserID nId,
32                 XnBool bSuccess, void* pCookie){
33   if (bSuccess){
34     printf("Usuario calibrado\n");
35     g_UserGenerator.GetSkeletonCap().StartTracking(nId);
36   } else {
37     printf("Fallo al calibrar al usuario %d\n", nId);
38    g_UserGenerator.GetPoseDetectionCap().
39      StartPoseDetection(POSE_TO_USE, nId);
40   }
41 }
```

Cuando el generador detecte un nuevo usuario en escena, se invocará al *callback User_NewUser*. En el anterior listado, puede verse cómo cuando se detecta un nuevo usuario por parte del generador, lo que se hará es esperar a identificar una determinada pose del mismo, de modo que esta pose sirva para comenzar el proceso de calibración (4-10). Por su parte, cuando el usuario desaparece de escena y se invoca al *callback User_LostUser* no se hace nada (12-14).

Una vez que se identifica la pose, se invocará el *callback* llamado *PoseDetected*, donde se detendrá el proceso de detección de pose –ya tenemos al usuario en la pose deseada– y se solicitará que comience el proceso de calibración (16-22).

Cuando el generador comience el proceso de calibración, éste invocará al *callback Calibration_Start*, que símplemente producirá un mensaje (24-28) por consola.

Cuando el proceso de calibración finalice, el generador invocará el *callback Calibration_End* donde, si todo ha ido bién, se indicará al generador que comience a rastrear el esqueleto del usuario con el método *StartTracking(nId)*. Sin embargo, si se ha producido un error en la callibración, se volverá a intentar identificar una pose preestablecida del usuario.

Los límites están en nuestra imaginación

Obviamente puede irse mucho más lejos de lo que se ha llegado con este sencillo ejemplo. Así, en la figura 31.17 puede verse la captura de una de las aplicaciones de ejemplo que acompañan OpenNI. En ella se superpone a la salida de los datos de un generador de profundidad, aquella producida por un generador de usuario del que se han leido los datos del esqueleto. El resultado muestra el esqueleto a partir de los puntos representativos identificados por el generador de usuario.

Piense ahora en lo que se puede conseguir empleando un motor como Ogre 3D. Los datos del esqueleto procedentes del generador de usuario, pueden conectarse al esqueleto de un personaje virtual o abatar. De este modo, las acciones del usuario tendrán una repercusión directa en las acciones del abatar. En la figura 31.18 puede verse una captura en la que un personaje dentro de un mundo virtual es manipulado directamente por un usuario. Fígese en la esquina inferior izquierda de la figura, donde se representa la silueta del usuario. El código de ejemplo puede descargarse libremente de https://github.com/ttair/TuxSinbad.

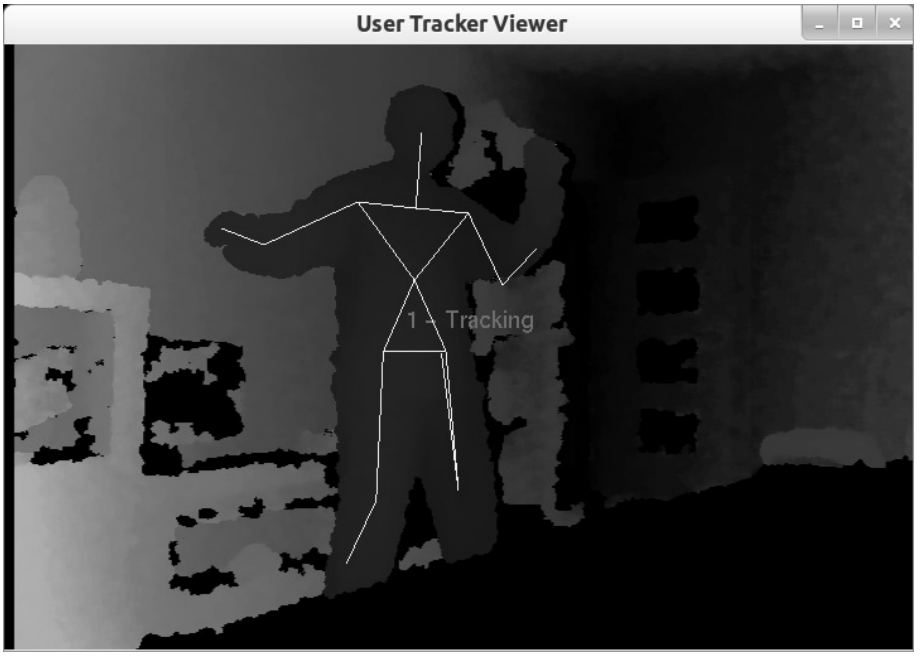


Figura 31.17: El esqueleto de una persona.

Figura 31.18: Integración en Ogre.

31.16. Realidad Aumentada

La Realidad Aumentada ofrece diversas posibilidades de interacción que pueden ser explotadas en diferentes ámbitos de aplicación, como en los videojuegos. En esta sección comenzaremos definiendo qué se entiende por Realidad Aumentada, así como algunas de las soluciones tecnológicas más empleadas.

Este nuevo paradigma de interacción se está utilizando en los últimos años en videojuegos comerciales, como veremos en la sección 31.16.1, con algunos títulos que han sido grandes éxitos a nivel mundial.

La Realidad Aumentada se encarga de estudiar las técnicas que permiten integrar en tiempo real contenido digital con el mundo real. Según la taxonomía descrita por Milgram y Kishino, los entornos de Realidad Mixta son aquellos en los que "se presentan objetos del mundo real y objetos virtuales de forma conjunta en una única pantalla". Esto abre un abanico de definiciones en la que se sitúan las aplicaciones de Realidad Aumentada (ver Figura 31.22).

A diferencia de la Realidad Virtual donde el usuario interactúa en un mundo totalmente virtual, la Realidad Aumentada se ocupa de generar capas de información virtual que deben ser correctamente alineadas con la imagen del mundo real para lograr una sensación de correcta integración.

El principal problema con el que deben tratar los sitemas de Realidad Aumentada es el denominado **registro**, que consiste en calcular la posición relativa de la cámara real respecto de la escena para poder generar imágenes virtuales correctamente alineadas con esa imagen real. Este registro debe ser preciso (errores de muy pocos milímetros son muy sensibles en determinadas aplicaciones, como en medicina o en soporte a las aplicaciones industriales) y robusto (debe funcionar correctamente en todo momento). En la figura 31.19 puede verse un ejemplo de una imagen obtenida con un registro correcto (centro) e incorrecto (abajo). Este registro puede realizarse empleando diferentes tipos de sensores y técnicas (las más extendidas son mediante el uso *tracking visual*).

Así, la Realidad Aumentada se sitúa entre medias de los entornos reales y los virtuales, encargándose de construir y alinear objetos virtuales que se integran en un escenario real.

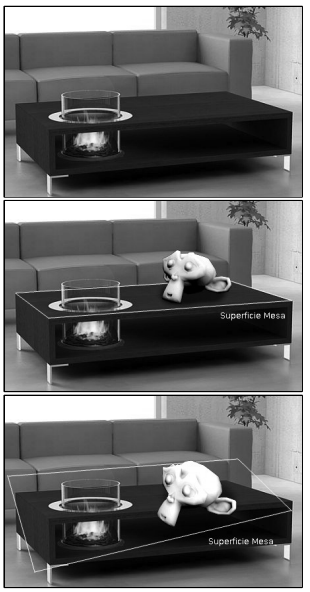


Figura 31.19: ***Arriba***: Escena real. El problema del registro trata de calcular la posición de la cámara real para poder dibujar objetos virtuales correctamente alineados. ***Centro***: Registro correcto. ***Abajo***: Registro incorrecto.

31.16.1. Un poco de historia

En esta breve introducción histórica nos centraremos en los principales hitos de la Realidad Aumentada que tienen relación directa con el mundo del desarrollo de videojuegos, aunque obviamente hay multitud de hitos importantes que deberían aparecer en una introducción histórica más general.

El primer sistema de Realidad Aumentada fue creado por Ivan Sutherland en 1968, empleando un casco de visión que permitía ver sencillos objetos 3D renderizados en *wireframe* en tiempo real. Empleaba dos sistemas de tracking para calcular el registro de la cámara; uno mecánico y otro basado en ultrasonidos (ver Figura 31.20).

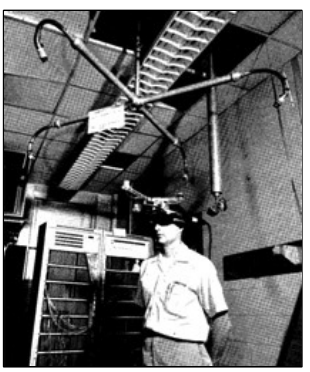


Figura 31.20: Primer Sistema de Realidad Aumentada de Sutherland.

Sin embargo no fue hasta 1992 cuando se acuñó el término de Realidad Aumentada por Tom Caudell y David Mizell, dos ingenieros de *Boeing* que proponían el uso de esta novedosa tecnología para mejorar la eficiencia de las tareas realizadas por operarios humanos asociadas a la fabricación de aviones.

En 1998, el ingeniero de Sony Jun Rekimoto crea un método para calcular completamente el tracking visual de la cámara (con 6 grados de libertad) empleando marcas 2D matriciales (códigos de barras cuadrados, ver Figura 31.21). Esta técnica sería la precursora de otros métodos de tracking visuales en los próximos años.

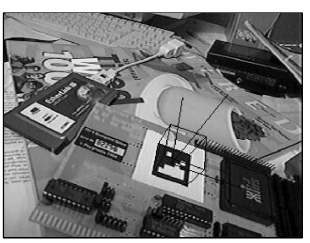


Figura 31.21: Marcas matriciales de Rekimoto.

Un año más tarde en 1999, Kato y Billinghurst presentan ARToolKit, una biblioteca de tracking visual de 6 grados de libertad que reconoce marcas cuadradas mediante patrones de reconocimiento. Debido a su liberación bajo licencia GPL se hace muy popular y es ampliamente utilizada en el ámbito de la Realidad Aumentada.

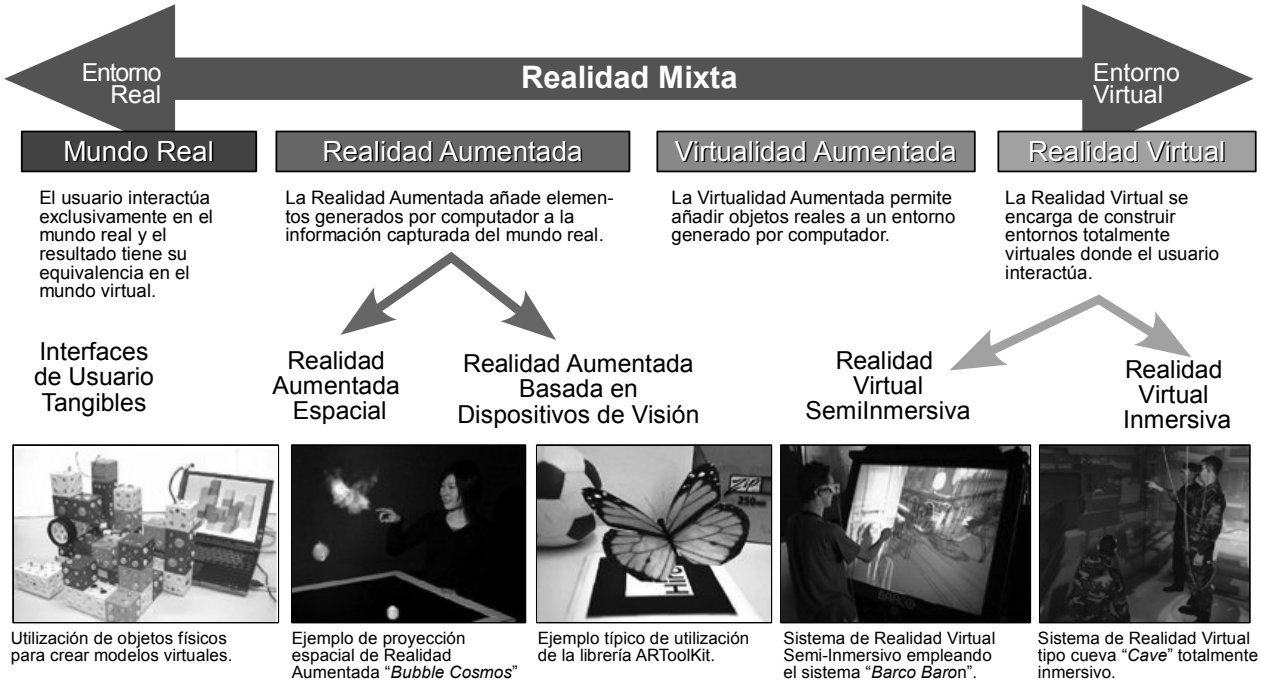


Figura 31.22: Taxonomía de Realidad Mixta según Milgram y Kishino.

Figura 31.23: AR-Quake.

Figura 31.24: H-Pacman.

En 2000, un grupo de investigadores de la *University of South Australia* presentan una extensión de Quake (AR-Quake, Figura 31.23) que permite jugar en primera persona en escenarios reales. El registro se realizaba empleando una brújula digital, un receptor de GPS y métodos de visión basados en marcas. Los jugadores debían llevar un sistema de cómputo portátil en una mochila, un casco de visión estereoscópica y un mando de dos botones.

En el 2003, Siemens lanza al mercado *Mozzies*, el primer juego de Realidad Aumentada para teléfonos móviles. El juego superpone mosquitos a la visión obtenida del mundo mediante una cámara integrada en el teléfono. Este juego fue premiado como el mejor videojuego para teléfonos móviles en dicho año.

En 2004 investigadores de la Universidad Nacional de Singapur presentan *Human Pacman*, un juego que emplea GPS y sistemas inerciales para registrar la posición de los jugadores. El PacMan y los fantasmas son en realidad jugadores humanos que corren por la ciudad portando ordenadores y sistemas de visión (Figura 31.24).

También en el 2004, la Universidad Técnica de Viena presenta el proyecto *Invisible Train* (ver Figura 31.26, el primer juego multi-usuario para PDAs. Esta aplicación se ejecutaba totalmente en las PDAs, sin necesidad de servidores adicionales para realizar procesamiento auxiliar. Los jugadores controlan trenes virtuales y deben intentar evitar que colisione con los trenes de otros jugadores. El proyecto utiliza la biblioteca *Studierstube* desarrollada en la misma universidad.

En 2005 A. Henrysson adapta la biblioteca ARToolKit para poder funcionar en Symbian, y crea un juego de Tenis (ver Figura 31.25) que gana un premio internacional el mismo año.

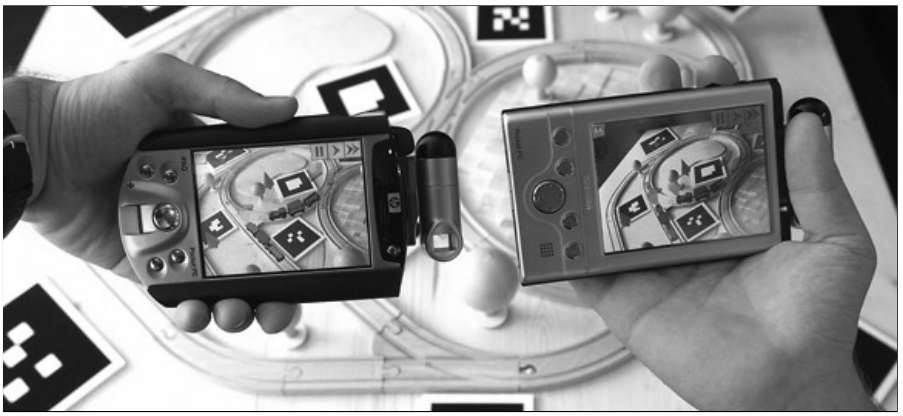


Figura 31.26: Interfaz de Invisible Train, de la Universidad de Viena.

El 2009 se presenta *ARhrrrr!* (ver Figura 31.27), el primer juego con contenido de alta calidad para smartphones con cámara. El teléfono utiliza la metáfora de ventana virtual para mostrar un mapa 3D donde disparar a Zombies y facilitar la salida a los humanos que están atrapados en él. El videojuego emplea de forma intensiva la GPU del teléfono delegando en la tarjeta todos los cálculos salvo el tracking basado en características naturales, que se realiza en la CPU. Así es posible distribuir el cálculo optimizando el tiempo final de ejecución.

El videojuego de PSP *Invizimals* (ver Figura 31.28), creado por el estudio español Novorama en 2009, alcanza una distribución en Europa en el primer trimestre de 2010 superior a las 350.000 copias, y más de 8 millones de copias a nivel mundial, situándose en lo más alto del ránking de ventas. Este juego emplea marcas para registrar la posición de la cámara empleando tracking visual.

Figura 31.25: AR-Tennis.

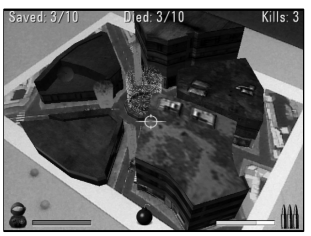


Figura 31.27: ARhrrrr!

Fotografía de ElPais.com 11-10-2009

Figura 31.28: Invizimals.

31.16.2. Características Generales

Según R. Azuma, un sistema de Realidad Aumentada debe cumplir las siguientes características:

1. **Combina mundo real y virtual**. El sistema incorpora información sintética a las imágenes percibidas del mundo real.
2. **Interactivo en tiempo real**. Así, los efectos especiales de películas que integran perfectamente imágenes 3D fotorrealistas con imagen real no se considera Realidad Aumentada porque no son calculadas de forma interactiva.
3. **Alineación 3D**. La información del mundo virtual debe ser tridimensional y debe estar correctamente alineada con la imagen del mundo real. Así, estrictamente hablando las aplicaciones que superponen capas gráficas 2D sobre la imagen del mundo real no son consideradas de Realidad Aumentada.

Siendo estrictos según la definición de Azuma, hay algunas aplicaciones que comercialmente se *venden* como de realidad aumentada que no deberían ser consideradas como tal, ya que el registro del mundo real no se realiza en 3D (como en el caso de *Wikitude* o *Layar* por ejemplo).

Figura 31.29: W. Ross Ashby (1903-1972), el padre de la Cibernética moderna.

Amplificación Inteligencia

El término **Amplificación de la Inteligencia** comenzó a utilizarse desde la *Introducción a la Cibernética* de William Ross Ashby, refiriéndose al uso de las tecnologías de la información para aumentar la inteligencia humana. También se denomina habitualmente *Aumentación Cognitiva*.

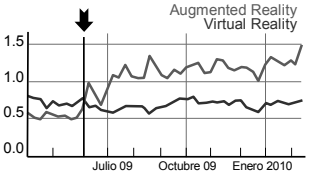


Figura 31.30: Porcentaje de búsquedas en Google por las cadenas *Augmented Reality* y *Virtual Reality*.

La combinación de estas tres características hacen que la Realidad Aumentada sea muy interesante para el usuario ya que complementa y mejora su visión e interacción del mundo real con información que puede resultarle extremadamente útil a la hora de realizar ciertas tareas. De hecho la Realidad Aumentada es considerada como una forma de *Amplificación de la Inteligencia* que emplea el computador para facilitar el trabajo al usuario.

La importancia de la Realidad Aumentada queda patente con el enorme interés que ha generado en los últimos meses. La prestigiosa publicación británicac *The Economist* aseguró en Septiembre de 2009 que *"intentar imaginar como se utilizará la Realidad Aumentada es como intentar predecir el futuro de la tecnología web en 1994"*.

Según la consultora *Juniper Research*, la Realidad Aumentada en dispositivos móviles generará más de 700 millones de dólares en el 2014, con más de 350 millones de terminales móviles con capacidad de ejecutar este tipo de aplicaciones.

Un indicador que puede ser significativo es la tendencia de búsqueda en la web. Desde Junio de 2009 la búsqueda por "*Augmented Reality*" supera el número de búsquedas realizadas con la cadena "*Virtual Reality*".

Una de las principales causas de este crecimiento en el uso de la Realidad Aumentada es debido a que mediante esta tecnología se amplían los espacios de interacción fuera del propio ordenador. Todo el mundo puede ser un interfaz empleando Realidad Aumentada sobre dispositivos móviles.

31.16.3. Alternativas tecnológicas

En el ámbito de la Realidad Aumentada existen varios *toolkits* que facilitan la construcción de aplicaciones. Sin embargo, para sacar el máximo partido a la tecnología es necesario dominar ciertos conceptos teóricos y de representación gráfica. La mayoría de sistemas están destinados a programadores con experiencia en desarrollo gráfico. A continuación enumeraremos algunas de las bibliotecas más famosas.

- **ARToolKit**: Es probablemente la biblioteca más famosa de Realidad Aumentada. Con interfaz en C y licencia libre permite desarrollar fácilmente aplicaciones de Realidad Aumentada. Se basa en marcadores cuadrados de color negro.
- **ARTag**: Es otra biblioteca con interfaz en C. Está inspirado en ARToolKit. El proyecto murió en el 2008, aunque es posible todavía conseguir el código fuente. El sistema de detección de marcas es mucho más robusto que el de ARToolKit.
- **OSGART**: Biblioteca en C++ que permite utilizar varias librerías de tracking (como ARToolKit, SSTT o Bazar).
- **FLARToolKit**: Implementación para Web (basada en Flash y ActionScript) del ARToolKit portado a Java *NyARToolKit*.
- **Otros ports de ARToolKit**: Existen multitud de versiones de ARToolKit portados en diferentes plataformas, como AndAR (para teléfonos Android), SLARToolkit, etc...

En esta sección nos basaremos en ARToolKit por ser actualmente el principal referente en tracking basado en marcas, y por existir multitud de versiones portadas a diferentes plataformas móviles.

31.17. ARToolKit

ARToolKit es una biblioteca de funciones para el desarrollo rápido de aplicaciones de Realidad Aumentada. Fue escrita originalmente en C por H. Kato, y mantenida por el HIT Lab de la Universidad de Washington, y el HIT Lab NZ de la Universidad de Canterbury (Nueva Zelanda).

ARToolKit facilita el problema del registro de la cámara empleando métodos de visión por computador, de forma que obtiene el posicionamiento relativo de 6 grados de libertad haciendo el seguimiento de marcadadores cuadrados en tiempo real, incluso en dispositivos de baja capacidad de cómputo. Algunas de las características más destacables son:

- **Tracking de una cámara**. ARToolKit en su versión básica soporta de forma nativa el tracking de una cámara, aunque puede utilizarse para tracking multicámara (si el programador se hace cargo de calcular el histórico de percepciones). La biblioteca soporta gran variedad de modelos de cámaras y modelos de color.
- **Marcas negras cuadradas**. Emplea métodos de tracking de superficies planas de 6 grados de libertad. Estas marcas pueden ser personalizadas, siempre que el patrón no sea simétrico en alguno de sus ejes.
- **Rápido y Multiplataforma**. Funciona en gran variedad de sistemas operativos (Linux, Mac, Windows, IRIX, SGI...), y ha sido portado a multitud de dispositivos portátiles y smartphones (Andorid, iPhone, PDAs...).
- **Comunidad Activa**. A través de los foros[3] y listas de correo se pueden resolver problemas particulares de uso de la biblioteca.
- **Licencia libre**. Esto permite utilizar, modificar y distribuir programas realizados con ARToolKit bajo la licencia GPL v2.

Figura 31.31: Uso de ARToolKit en un dispositivo de visión estereo.

31.17.1. Instalación y configuración

En esencia ARToolKit es una biblioteca de funciones para el desarrollo de aplicaciones de Realidad Aumentada, que utiliza a su vez otras bibliotecas. En Debian necesitaremos instalar los siguientes paquetes:

```
# apt-get install freeglut3-dev libgl1-mesa-dev libglu1-mesa-dev
libxi-dev libxmu-dev libjpeg-dev
```

A continuación ejecutamos `./Configure` para obtener un *Makefile* adaptado a nuestro sistema. Elegimos Video4Linux2 en el driver de captura de video porque disponemos de una cámara integrada de este tipo (la versión de ARToolKit con la que vamos a trabajar está parcheada para soportar este tipo de dispositivos), en la segunda pregunta no utilizaremos las instrucciones de ensamblador en ccvt (por disponer de una arquitectura ia64), habilitaremos los símbolos de depuración, y activaremos el soporte hardware para `GL_NV_texture_rectangle`, ya que la tarjeta gráfica del equipo los soporta:

```
carlos@kurt:ARToolKit$ ./Configure
Select a video capture driver.
  1: Video4Linux
  2: Video4Linux+JPEG Decompression (EyeToy)
  3: Video4Linux2
```

[3]Foros de ARToolKit: `http://www.hitlabnz.org/wiki/Forum`

Figura 31.32: Aunque las bibliotecas con las que trabajaremos en este documento son libres, sólo describiremos el proceso de instalación y configuración bajo GNU/Linux. *Imagen original de sfgate.com.*

```
  4: Digital Video Camcoder through IEEE 1394 (DV Format)
  5: Digital Video Camera through IEEE 1394 (VGA NC Image Format)
  6: GStreamer Media Framework
Enter : 3

Color conversion should use x86 assembly (not working for 64bit)?
Enter : n
Do you want to create debug symbols? (y or n)
Enter : y
Build gsub libraries with texture rectangle support? (y or n)
GL_NV_texture_rectangle is supported on most NVidia graphics cards
and on ATi Radeon and better graphics cards
Enter : y
  create ./Makefile
  create lib/SRC/Makefile
  ...
  create include/AR/config.h
Done.
```

Finalmente compilamos las bibliotecas desde el mismo directorio, ejecutando `make`. Si todo ha ido bien, ya tenemos compiladas las bibliotecas de ARToolKit. Estas bibliotecas no requieren estar instaladas en ningún directorio especial del sistema, ya que se compilan como bibliotecas estáticas, de forma que están incluidas en cada ejecutable que se construye. Los ficheros `makefile` que utilizaremos en los ejemplos tendrán definido un camino (relativo o absoluto) hasta la localización en disco de estas bibliotecas. A continuación veremos un ejemplo de funcionamiento básico.

31.18. El esperado “Hola Mundo!”

Aunque todavía quedan muchos aspectos que estudiar, comenzaremos con una aplicación mínima que dibuja una tetera 3D localizada en el centro de una marca de ARToolKit. El siguiente listado muestra el código completo (¡menos de 110 líneas!) del *Hola Mundo!*. Al final de la sección se mostrará el listado del `makefile` necesario para compilar este ejemplo.

Figura 31.33: La impresionante salida en pantalla del *“Hola Mundo”* de Realidad Aumentada.

Listado 31.24: “Hola Mundo!” con ARToolKit.

```
 1 #include <GL/glut.h>
 2 #include <AR/gsub.h>
 3 #include <AR/video.h>
 4 #include <AR/param.h>
 5 #include <AR/ar.h>
 6 // ==== Definicion de variables globales ==========================
 7 int    patt_id;            // Identificador unico de la marca
 8 double patt_trans[3][4];   // Matriz de transformacion de la marca
 9 // ======== print_error ================================================
10 void print_error (char *error) {  printf(error); exit(0); }
11 // ======== cleanup ================================================
12 static void cleanup(void) {
13   arVideoCapStop();            // Libera recursos al salir ...
14   arVideoClose();
15   argCleanup();
16 }
17 // ======== draw ====================================================
18 static void draw( void ) {
19   double  gl_para[16];   // Esta matriz 4x4 es la usada por OpenGL
20   GLfloat mat_ambient[]     = {0.0, 0.0, 1.0, 1.0};
21   GLfloat light_position[]  = {100.0,-200.0,200.0,0.0};
22
23   argDrawMode3D();              // Cambiamos el contexto a 3D
24   argDraw3dCamera(0, 0);        // Y la vista de la camara a 3D
25   glClear(GL_DEPTH_BUFFER_BIT); // Limpiamos buffer de profundidad
26   glEnable(GL_DEPTH_TEST);
```

```
27    glDepthFunc(GL_LEQUAL);
28
29    argConvGlpara(patt_trans, gl_para);   // Convertimos la matriz de
30    glMatrixMode(GL_MODELVIEW);           // la marca para ser usada
31    glLoadMatrixd(gl_para);               // por OpenGL
32    // Esta ultima parte del codigo es para dibujar el objeto 3D
33    glEnable(GL_LIGHTING);  glEnable(GL_LIGHT0);
34    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
35    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
36      glTranslatef(0.0, 0.0, 60.0);
37      glRotatef(90.0, 1.0, 0.0, 0.0);
38      glutSolidTeapot(80.0);
39    glDiable(GL_DEPTH_TEST);
40  }
41  // ======== init ================================================
42  static void init( void ) {
43    ARParam  wparam, cparam;  // Parametros intrinsecos de la camara
44    int xsize, ysize;         // Tamano del video de camara (pixels)
45
46    // Abrimos dispositivo de video
47    if(arVideoOpen("") < 0) exit(0);
48    if(arVideoInqSize(&xsize, &ysize) < 0) exit(0);
49
50    // Cargamos los parametros intrinsecos de la camara
51    if(arParamLoad("data/camera_para.dat", 1, &wparam) < 0)
52      print_error ("Error en carga de parametros de camara\n");
53
54    arParamChangeSize(&wparam, xsize, ysize, &cparam);
55    arInitCparam(&cparam);   // Inicializamos la camara con çparam"
56
57    // Cargamos la marca que vamos a reconocer en este ejemplo
58    if((patt_id=arLoadPatt("data/simple.patt")) < 0)
59      print_error ("Error en carga de patron\n");
60
61    argInit(&cparam, 1.0, 0, 0, 0, 0);   // Abrimos la ventana
62  }
63  // ======== mainLoop ============================================
64  static void mainLoop(void) {
65    ARUint8 *dataPtr;
66    ARMarkerInfo *marker_info;
67    int marker_num, j, k;
68
69    double p_width     = 120.0;        // Ancho del patron (marca)
70    double p_center[2] = {0.0, 0.0};   // Centro del patron (marca)
71
72    // Capturamos un frame de la camara de video
73    if((dataPtr = (ARUint8 *)arVideoGetImage()) == NULL) {
74      // Si devuelve NULL es porque no hay un nuevo frame listo
75      arUtilSleep(2);  return;  // Dormimos el hilo 2ms y salimos
76    }
77    argDrawMode2D();
78    argDispImage(dataPtr, 0,0);    // Dibujamos lo que ve la camara
79
80    // Detectamos la marca en el frame capturado (ret -1 si error)
81    if(arDetectMarker(dataPtr, 100, &marker_info, &marker_num) < 0){
82      cleanup(); exit(0);   // Si devolvio -1, salimos del programa!
83    }
84    arVideoCapNext();    // Frame pintado y analizado... A por otro!
85    // Vemos donde detecta el patron con mayor fiabilidad
86    for(j = 0, k = -1; j < marker_num; j++) {
87      if(patt_id == marker_info[j].id) {
88        if (k == -1) k = j;
89        else if(marker_info[k].cf < marker_info[j].cf) k = j;
90      }
91    }
92    if(k != -1) {   // Si ha detectado el patron en algun sitio...
93      // Obtenemos transformacion entre la marca y la camara real
94      arGetTransMat(&marker_info[k], p_center, p_width, patt_trans);
95      draw();       // Dibujamos los objetos de la escena
96    }
97    argSwapBuffers(); // Cambiamos el buffer con lo que dibujado
```

```
 98 }
 99 // ======== Main ================================================
100 int main(int argc, char **argv) {
101   glutInit(&argc, argv);     // Creamos la ventana OpenGL con Glut
102   init();                    // Llamada a nuestra funcion de inicio
103   arVideoCapStart();         // Creamos un hilo para captura video
104   argMainLoop( NULL, NULL, mainLoop );   // Asociamos callbacks...
105   return (0);
106 }
```

El ciclo de desarrollo puede resumirse en tres grandes etapas: 1. **Inicialización**: Consiste en leer los parámetros asociados a la cámara y la descripción de las marcas que se van a utilizar. 2. **Bucle Principal** (*Main Loop*): Es la etapa principal y está formada por un conjunto de subetapas que veremos a continuación. 3. **Finalización**: Libera los recursos requeridos por la aplicación.

La etapa del Bucle Principal está formada por 4 subetapas funcionales que se realizan repetidamente hasta que el usuario decide finalizar la aplicación:

1. **Captura**. Se obtiene un *frame* de la cámara de vídeo. En el *"Hola Mundo!"* se realiza llamando a la función `arVideoGetImage` en la línea 73.
2. **Detección**. Se identifican las marcas en el *frame* anterior. En el ejemplo se llama a la función `arDetectMarker` en la línea 81.
3. **Transformación**. Se calcula la posición relativa entre las marcas detectadas y la cámara física. Se realiza llamando a la función `arGetTransMat` en la línea 94.
4. **Dibujado**. Se dibujan los objetos virtuales situando la cámara virtual en la posición relativa anteriormente calculada. En el *Hola Mundo* se ha creado una función propia `draw` que es llamada desde el mainLoop en la línea 95.

Marcas Personalizadas

ARToolKit permite crear fácilmente marcadores propios. En este primer ejemplo utilizaremos una marca previamente entrenada.

En los ejemplos de este documento nos basaremos en el uso de las bibliotecas GLUT (*OpenGL Utility Toolkit*) que nos facilitan el desarrollo de aplicaciones OpenGL proporcionando sencillas funciones de *callback* para el manejo de eventos de teclado y ratón, así como la fácil apertura de ventanas multiplataforma. Estas bibliotecas están pensadas para el aprendizaje de OpenGL, y para la construcción de pequeñas aplicaciones. Para facilitar la escritura de código en estos primeros ejemplos se utilizarán algunas variables globales (por la imposibilidad de pasar parámetros adicionales a las funciones de callback de GLUT).

En un primer estudio, se pueden identificar algunos de los bloques funcionales descritos anteriormente. Por ejemplo, en las líneas 43-61 se encuentra definido el bloque de inicialización (función `init`), que es llamado desde `main` (línea 102). El bucle principal está definido en las líneas 65-97 y la liberación de recursos se realiza con la llamada a funciones propias de ARToolKit en las líneas 13-15.

En la última sección del capítulo se estudiará la integración de ARToolKit con OpenCV y Ogre, creando clases específicas para el tracking y la gestión de vídeo.

Uso de **/dev/video***

Si tuviéramos dos cámaras en el sistema, y quisiéramos abrir la segunda (con interfaz V4L) llamaríamos a la función *arVideoOpen* con la cadena *"-dev=/dev/video1"*. Podemos ver una descripción completa de los parámetros soportados por esta función en el directorio de documentación *doc/video/* de ARToolKit.

31.18.1. Inicialización

El bloque de inicialización (función `init`), comienza abriendo el dispositivo de video con `arVideoOpen`. Esta función admite parámetros de configuración en una cadena de caracteres.

A continuación, línea 48, se obtiene el tamaño de la fuente de vídeo mediante `arVideoInqSize`. Esta función escribe en las direcciones de memoria reservadas para dos enteros el ancho y alto del vídeo. En las siguientes líneas cargamos los parámetros intrínsecos de la cámara obtenidos en la etapa de calibración. En este ejemplo utilizaremos un fichero genérico *camera_para.dat* válido para la mayoría de webcams. Sin embargo, para obtener resultados más precisos, lo ideal es trabajar con un fichero adaptado a nuestro modelo de cámara concreto. Veremos cómo crear este fichero específico para cada cámara posteriormente.

Así, con la llamada a `arParamLoad` se rellena la estructura `ARParam` especificada como tercer parámetro con las características de la cámara. Estas características son independientes de la resolución a la que esté trabajando la cámara, por lo que tenemos que instanciar los parámetros concretamente a la resolución en píxeles. Para ello, se utiliza la llamada a `arParamChangeSize` (línea 54) especificando la resolución concreta con la que trabajará la cámara en este ejemplo. Una vez obtenidos los parámetros de la cámara instanciados al caso concreto de este ejemplo, se cargan en las estructuras de datos internas de ARToolKit, mediante la llamada a `arInitCparam`.

En la última parte cargamos el patrón asociado a la marca (línea 58). Finalmente abrimos la ventana de OpenGL (mediante la biblioteca auxiliar Gsub de ARToolKit) `argInit` en la línea 61, pasándole como primer parámetro la configuración de la cámara. El segundo parámetro indica el factor de *zoom* (en este caso, sin zoom).

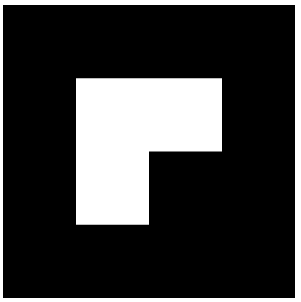


Figura 31.34: Aunque es posible definir marcas personalizadas, las que ofrecen mejores resultados son las basadas en patrones sencillos, como la empleada en este ejemplo.

31.18.2. Bucle Principal

El primer paso a realizar en el bucle principal es recuperar un *frame* de la cámara de vídeo. Mediante la función `arVideoGetImage` obtenemos una imagen (la llamada devuelve un puntero a un buffer donde se encuentra la imagen capturada). La llamada devuelve `NULL` si no hay una nueva imagen (en el caso de que llamemos de nuevo muy pronto; con mayor frecuencia que la soportada por la cámara). Si esto ocurre, simplemente dormimos el hilo 2ms (línea 75)y volvemos a ejecutar el *mainLoop*.

A continuación dibujamos en la ventana (en modo 2D, línea 77) el buffer que acabamos de recuperar de la cámara (línea 78).

La llamada a `arDetectMarker` localiza las marcas en el buffer de entrada. En la línea 81 el segundo parámetro de valor 100 se corresponde con el valor umbral de binarización (a blanco y negro, ver Figura 31.35) de la imagen. El propósito de este valor umbral (*threshold*) se explicará en detalle más adelante. Esta función nos devuelve en el tercer parámetro un puntero a una lista de estructuras de tipo `ARMarkerInfo`, que contienen información sobre las marcas detectadas (junto con un grado de fiabilidad de la detección), y como cuarto parámetro el número de marcas detectadas.

De esta forma, ARToolKit nos devuelve *"posibles"* posiciones para cada una de las marcas detectas. Incluso cuando estamos trabajando con una única marca, es común que sea detectada en diferentes posiciones (por ejemplo, si hay algo parecido a un cuadrado negro en la escena). ¿Cómo elegimos la correcta en el caso de que tengamos varias detecciones? ARToolKit asocia a cada percepción una probabilidad

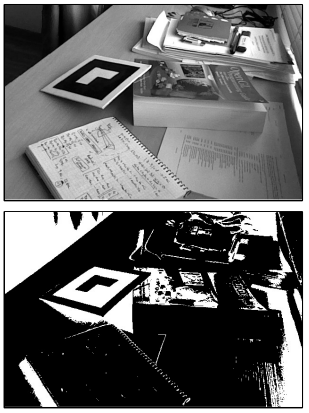


Figura 31.35: Ejemplo de proceso de binarización de la imagen de entrada para la detección de marcas.

de que lo percibido sea una marca, en el campo `cf` (*confidence value*). En la tabla 31.1 se muestra la descripción completa de los campos de esta estructura. Como se puede comprobar, todos los campos de la estructura ARMarkerInfo se refieren a coordenadas 2D, por lo que aún no se ha calculado la posición relativa de la marca con la cámara.

Así, en las líneas 86–91 guardamos en la variable `k` el índice de la lista de marcas detectadas aquella percepción que tenga mayor probabilidad de ser la marca (cuyo valor de fiabilidad sea mayor).

Mediante la llamada a `arGetTransMat` (línea 94) obtenemos en una matriz la transformación relativa entre la marca y la cámara (matriz 3x4 de doubles); es decir, obtiene la posición y rotación de la cámara con respecto de la marca detectada. Para ello es necesario especificar el centro de la marca, y el ancho. Esta matriz será finalmente convertida al formato de matriz homogenea de 16 componentes utilizada por OpenGL mediante la llamada a `argConvGlpara` en la línea 29.

Cuadro 31.1: Campos de la estructura *ARMarkerInfo*

Tipo	Campo	Descripción
`int`	**`area`**	Tamaño en píxeles de la región detectada.
`int`	**`id`**	Identificador (único) de la marca.
`int`	**`dir`**	Dirección. Codifica mediante un valor numérico (0..3) la rotación de la marca detectada. Cada marca puede tener 4 rotaciones distintas.
`double`	**`cf`**	Valor de confianza. Probabilidad de ser una marca (entre 0 y 1).
`double`	**`pos[2]`**	Centro de la marca (en espacio 2D).
`double`	**`line[4][3]`**	Ecuaciones de las 4 aristas de la marca. Las aristas se definen con 3 valores (a,b,c), empleando la ecuación implícita de la recta $ax + by + c = 0$.
`double`	**`vertex[4][2]`**	Posición de los 4 vértices de la marca (en espacio 2D).

31.18.3. Finalización y función Main

En la función `cleanup` se liberan los recursos al salir de la aplicación. Se hace uso de funciones de ARToolKit para detener la cámara de vídeo, y limpiar las estructuras de datos internas de ARToolKit.

En la función `main` se registran los *callbacks* en la línea 104 mediante la función `argMainLoop`. En este ejemplo, se pasa como primer y segundo parámetro `NULL` (correspondientes a los manejadores de ratón y teclado respectivamente). Por su parte, se asocia la función que se estará llamando constantemente en el bucle principal. En el caso de este ejemplo se llama `mainLoop`.

31.19. Las Entrañas de ARToolKit

En este apartado veremos algunos detalles sobre cómo funciona internamente *ARToolKit*. ARToolKit está formado por tres módulos (ver Figura 31.36):

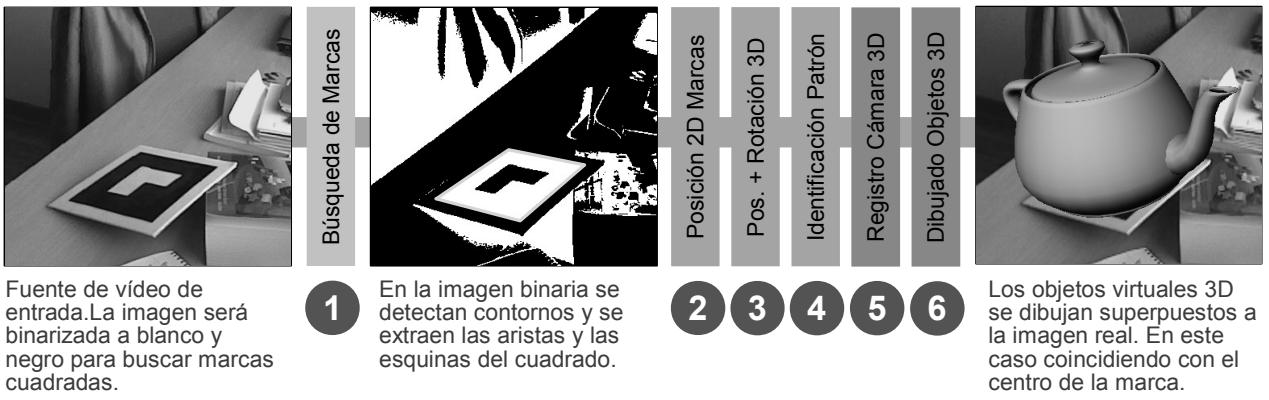


Figura 31.37: Esquema funcional de ARToolKit.

- **Video**: Este módulo contiene las funciones para obtener frames de vídeo de los dispositivos soportados por el Sistema Operativo. El prototipo de las funciones de este módulo se encuentran el fichero de cabecera `video.h`.
- **AR**: Este módulo principal contiene las funciones principales de *tracking* de marcas, calibración y estructuras de datos requeridas por estos métodos. Los ficheros de cabecera `ar.h`, `arMulti.h` (subrutinas para gestión multi-patrón) y `param.h` describen las funciones asociadas a este módulo.
- **Gsub y Gsub_Lite**: Estos módulos contienen las funciones relacionadas con la etapa de representación. Ambas utilizan GLUT, aunque la versión "_Lite" es más eficiente y no tiene dependencias con ningún sistema de ventanas concreto. En estos módulos se describen los ficheros de cabecera `gsub.h`, `gsub_lite.h` y `gsubUtil.h`.

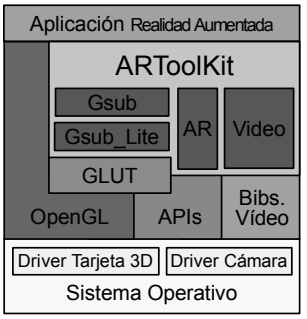


Figura 31.36: Arquitectura de ARToolKit.

Estos módulos están totalmente desacoplados, de forma que es posible utilizar ARToolKit sin necesidad de emplear los métodos de captura de vídeo del primer módulo, o sin emplear los módulos de representación *Gsub* o *Gsub_Lite*, como veremos en la posterior integración con Ogre y OpenCV.

31.19.1. Principios Básicos

ARToolKit está basado en un algoritmo de detección de bordes y un método rápido de estimación de la orientación. La figura 31.37 resume el principio básico de funcionamiento de ARToolKit. Inicialmente las funciones de ARToolKit nos aislan de la complejidad de tratar con diferentes dispositivos de vídeo, por lo que la captura del *frame* actual es una simple llamada a función. Con esta imagen se inicia el primer paso de búsqueda de marcas. La imagen se convierte a blanco y negro para facilitar la detección de formas cuadradas (en realidad este paso se realiza en dos etapas (ver Figura 31.38); primero se convierte a escala de grises (b), y después se binariza (c) eligiendo un parámetro de umbral "*threshold*" que elige a partir de qué valor de gris (de entre 256 valores distintos) se considera blanco o negro (ver Figura 31.39).

A continuación el algoritmo de visión por computador extrae componentes conectados de la imagen previamente binarizada, Figura 31.38 (d), cuya area es suficientemente grande como para detectar una marca. A estas regiones se les aplica un rápido algoritmo de detección de contornos (e), obteniendo a continuación los vértices y aristas que definen la región de la marca en 2D (f).

Figura 31.39: Ejemplo de binarización con diferentes valores de *Threshold*.

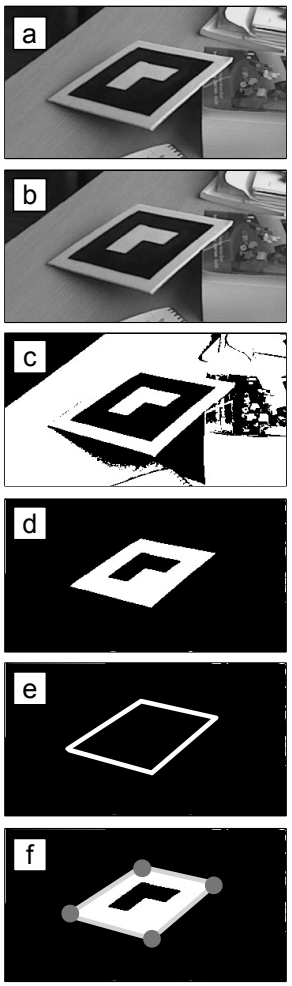


Figura 31.38: Pasos seguidos por ARToolKit para la detección e identificación de marcas.

Con la región definida, se procede a una fase de normalización en la que se extrae el *contenido* de la marca (la zona central que la diferencia de otras marcas) y se compara con los patrones de las marcas conocidas (etapa 4 de la Figura 31.37).

Conociendo las posiciones 2D de las aristas y vértices que definen el marcador 2D, y el modelo de proyección de la cámara es posible estimar la posición y rotación 3D de la cámara relativamente a la marca.

En el módulo 2 estudiamos qué suponía el cálculo de la posición y la rotación de la cámara en términos de transformaciones en OpenGL. El uso de marcas cuadradas de un tamaño previamente conocido nos permite definir un sistema de coordenadas local a cada marca, de modo que empleando métodos de visión por computador obtengamos la matriz de transformación 4x4 del sistema de coordenadas de la marca al sistema de coordenadas de la cámara T_{cm} (Ecuación 31.1).

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & T_x \\ R_{21} & R_{22} & R_{23} & T_y \\ R_{31} & R_{32} & R_{33} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_m \\ Y_m \\ Z_m \\ 1 \end{bmatrix} = T_{cm} \times \begin{bmatrix} X_m \\ Y_m \\ Z_m \\ 1 \end{bmatrix} \tag{31.1}$$

De este modo, conociendo la proyección de cada esquina de la marca (e) sobre las coordenadas de pantalla (xc, yc), y las restricciones asociadas al tamaño de las marcas y su geometría cuadrada, es posible calcular la matriz T_{cm}. La Figura 31.40 resume esta transformación.

El cálculo aproximado de la matriz de transformación que representa la rotación R y la traslación T desde las coordenadas de la marca a las coordenadas de la cámara se denominan en inglés *pose estimation* y *position estimation*.

Finalmente es posible dibujar cualquier objeto 3D correctamente alineado con la escena. Si conocemos la posición absoluta de las marcas respecto de algún sistema de coordenadas global, es posible representar los objetos posicionados globalmente. Veremos más detalles sobre el posicionamiento global en los próximos capítulos.

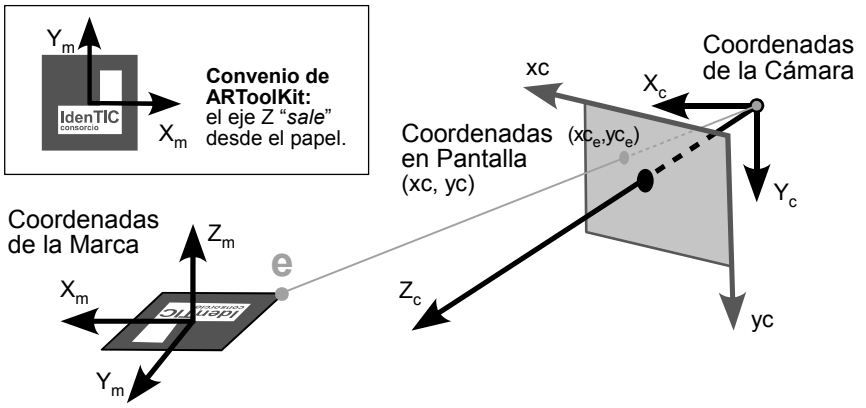


Figura 31.40: Sistema de coordenadas de ARToolKit

31.19.2. Calibración de la Cámara

Como vimos en el ejemplo del *"Hola Mundo!"*, los parámetros de la cámara se cargan en tiempo de ejecución de un archivo, mediante la llamada a la función `arParamLoad` (ver Listado del ejemplo). Aunque en el primer ejemplo trabajamos con un fichero de descripcion genérico de cámaras (válido para multitud de webcams), es preferible realizar un proceso previo de calibración que obtenga los parámetros intrínsecos de la cámara.

ARToolKit proporciona dos etapas de calibración; la primera (que es la que estudiaremos en esta sección) nos permite obtener un fichero de cámara válido para la mayoría de aplicaciones que contiene información sobre el **punto central de la imagen** y el **factor de distorsión** de las lentes. Si se quiere una mayor precisión (que contenga la **distancia focal** de la cámara), es necesario realizar la segunda etapa (ver documentación oficial de ARToolKit).

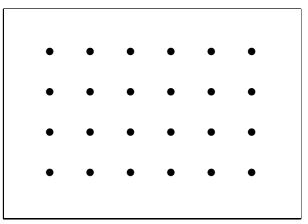


Figura 31.41: Patrón de calibración de la cámara.

Para realizar la calibración de una etapa de la cámara, necesitaremos imprimir el patrón de 24 puntos separados entre sí 4cm (ver Figura 31.41). Hecho esto, ejecutaremos la aplicación `calib_camera2` de la distribución de ARToolKit. La aplicación nos pedirá la longitud en milímetros entre cada punto del patrón de calibración (deberá ser 40 si hemos imprimido el patrón a tamaño 1:1, en otro caso mediremos el espacio físico entre el centro de los puntos).

```
carlos@kurt:calib_camera2$ ./calib_camera2
Input the distance between each marker dot, in millimeters: 40
-----------
Press mouse button to grab first image,
or press right mouse button or [esc] to quit.
```

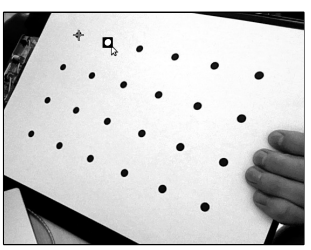


Figura 31.42: Marcado de círculos del patrón.

Hecho esto nos aparecerá una ventana con la imagen que percibe la cámara. Moveremos el patrón para que todos los puntos aparezcan en la imagen y presionaremos el botón izquierdo del ratón una vez sobre la ventana para congelar la imagen. Ahora tenemos que definir un rectángulo que rodee cada círculo del patrón (ver Figura 31.42 empleando el siguiente orden: primero el círculo más cercano a la esquina superior izquierda, y a continuación los de su misma fila. Luego los de la segunda fila comenzando por el de la izquierda y así sucesivamente. Es decir, los círculos del patrón deben ser recorridos en orden indicado en la Figura 31.43.

El programa marcará una pequeña cruz en el centro de cada círculo que hayamos marcado (ver Figura 31.42), y aparecerá una línea indicando que ha sido señalada como se muestra a continuación. Si no aparece una cruz en rojo es porque el círculo no se ha detectado y tendrá que ser de nuevo señalado.

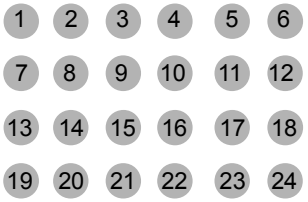


Figura 31.43: Orden de marcado de los círculos del patrón de calibración.

```
Grabbed image 1.
-----------
Press mouse button and drag mouse to rubber-bound features (6 x 4),
or press right mouse button or [esc] to cancel rubber-bounding & retry
     grabbing.
Marked feature position   1 of  24
Marked feature position   2 of  24
Marked feature position   3 of  24
...
Marked feature position  24 of  24
-----------
Press mouse button to save feature positions,
or press right mouse button or [esc] to discard feature positions &
    retry grabbing.
```

Una vez que se hayan marcado los 24 puntos, se pulsa de nuevo el botón izquierdo del ratón sobre la imagen. Esto almacenará la posición de las marcas para la primera imagen, y descongelará la cámara, obteniendo una salida en el terminal como la siguiente.

Precisión en calibración

Como es obvio, a mayor número de imágenes capturadas y marcadas, mayor precisión en el proceso de calibración. Normalmente con 5 ó 6 imágenes distintas suele ser suficiente.

```
### Image no.1 ###
 1,  1: 239.50, 166.00
 2,  1: 289.00, 167.00
 ...
 6,  4: 514.00, 253.50
-----------
Press mouse button to grab next image,
or press right mouse button or [esc] to calculate distortion param.
```

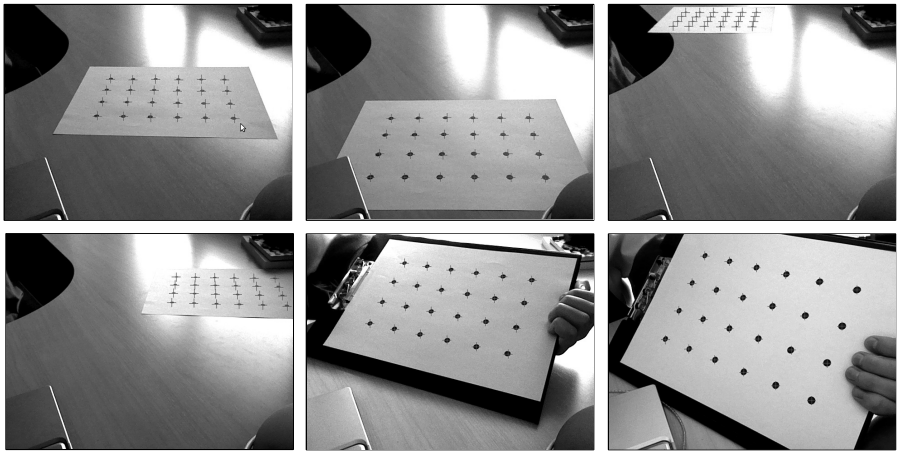


Figura 31.44: Ejemplo de posicionamiento de patrones de calibración.

Como se indica en el manual de ARToolKit, es necesario capturar entre 5 y 10 imágenes siguiendo el mismo proceso, variando el ángulo y la posición en la que se presenta el patrón de calibración. En la Figura 31.44 se muestran 6 ejemplos de diferentes imágenes capturadas para la calibración de una cámara.

Cuando te haya realizado un número de capturas adecuado pulsaremos el botón derecho del ratón (o la tecla `ESC`) para calcular el parámetro de distorsión. En el terminal aparecerá una salida como la siguiente, indicando al final el centro (X,Y) de la cámara, y el factor de distorsión. El cálculo de estos parámetros puede requerir varios segundos.

```
Press mouse button to grab next image,
or press right mouse button or [esc] to calculate distortion param.
[320.0, 240.0, -13.5] 0.459403
[370.0, 190.0, -15.3] 0.457091
...
[375.0, 211.0, -16.4] 0.456635
--------------
Center X: 375.000000
       Y: 211.000000
Dist Factor: -16.400000
Size Adjust: 0.978400
--------------
```

A continuación la aplicación nos muestra el resultado del cálculo de estos parámetros, para que el usuario realice una comprobación visual. Cada vez que pulsemos con el botón izquierdo del ratón se mostrará una imagen con líneas de color rojo que deben pasar por el centro de cada círculo de calibración (como las dos imágenes mostradas en la Figura 31.45).

```
Checking fit on image   1 of   6.
Press mouse button to check fit of next image.
```

Finalmente, tras aceptar los resultados mostrados por la aplicación, se calcularán todos los parámetros que se guardarán en el fichero que le indiquemos al final del proceso.

Este nuevo fichero creado será el que le indicaremos a ARToolKit que utilice en la llamada a la función `arParamLoad`.

```
-- loop:-50 --
F = (816.72,746.92), Center = (325.0,161.0): err = 0.843755
-- loop:-49 --
F = (816.47,747.72), Center = (325.0,162.0): err = 0.830948
...
Calibration succeeded. Enter filename to save camera parameter.
-------------------------------------
SIZE = 640, 480
Distortion factor = 375.000000 211.000000 -16.400000 0.978400
770.43632 0.00000 329.00000 0.00000
0.00000 738.93605 207.00000 0.00000
0.00000 0.00000 1.00000 0.00000
-------------------------------------
Filename: logitech_usb.dat
```

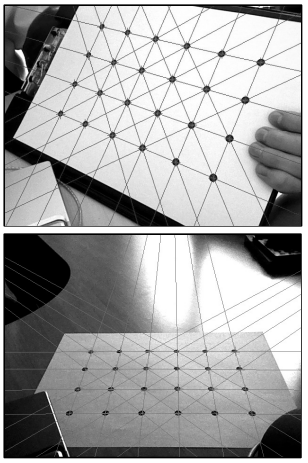


Figura 31.45: Dos imágenes resultado del cálculo del centro y el factor de distorsión.

31.19.3. Detección de Marcas

En la sección 31.19.1 hemos visto que ARToolKit extrae los vértices que definen las cuatro esquinas de una marca. En la etapa de detección de la marca, es necesario además identificar qué patrón está siendo detectado. Para ello ARToolKit primero normaliza la imagen detectada (eliminando la distorsión debida a la perspectiva) para posteriormente comparar el patrón con las plantillas de los patrones que puede reconocer la aplicación (aquellas que hayan sido cargadas con la función `arLoadPatt`).

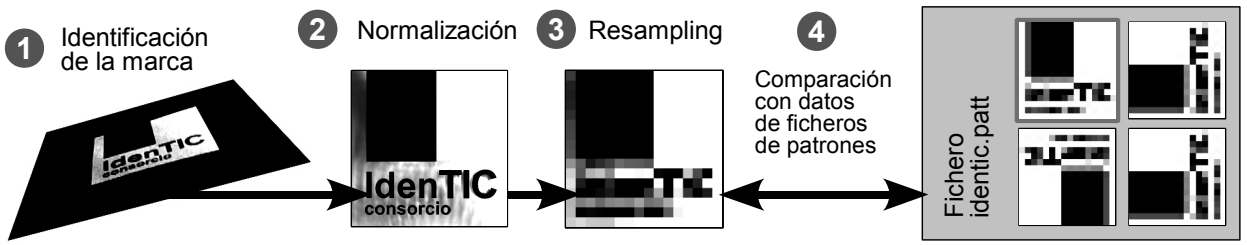


Figura 31.46: Pasos para la identificación de patrones.

El proceso de normalización de los patrones requiere mucho tiempo, por lo que es un problema cuando se utilizan muchos marcadores. Tras la normalización, se reduce la resolución del patrón normalizado antes de pasar a la fase de comparación. Cuanto mayor es la resolución del patrón reescalado, mayor es la precisión de ARToolKit, pero require más capacidad de cómputo para realizar las operaciones.

En el fichero `config.h` de ARToolKit pueden defirse algunos parámetros relacionados con la detección de marcas. A continuación se indican los valores que trae por defecto la distribución de ARToolKit:

- `#define AR_SQUARE_MAX 50`: Este parámetro define el número máximo de marcadores que serán detectados en cada imagen.
- `#define AR_PATT_NUM_MAX 50`: Número máximo de patrones que pueden ser cargados por el usuario.
- `#define AR_PATT_SIZE_X 16`: Resolución (número de píxeles) en horizontal del patrón cuando es resampleado.
- `#define AR_PATT_SIZE_Y 16`: Resolución (número de píxeles) en horizontal del patrón cuando es resampleado.
- `#define AR_PATT_SAMPLE_NUM 64`: Número máximo de pasos empleados para resamplear el patrón.

De esta forma, el patrón reconocido incialmente es normalizado y resampleado para obtener una imagen como se muestra en la Figura 31.46. Por defecto, esta representación se realizará en 64 pasos, generando una imagen de 16x16 píxeles. Esta matriz de 16x16 píxeles se compará con los datos contenidos en los ficheros de patrones. Estos ficheros simplemente contienen 4 matrices con el valor de gris de cada píxel del patrón. Cada matriz se corresponde con el patrón rotado 90^o (ver Figura 31.46). Podemos abrir cualquier fichero .patt (como el que utilizamos en el *Hola Mundo!*) y veremos los valores en ASCII correspondientes a las 4 matrices.

Limitaciones

Si utilizamos únicamente métodos de registro basados en marcas como en el caso de ARToolKit, la principal limitación viene cuando ninguna marca es visible en un momento dado, ya que no es posible obtener la posición de la cámara virtual. Además, si se oculta alguna parte de la marca, el método de detección, tal y como hemos visto, no será capaz de identificar las 4 esquinas de la región y el método de identificación fallará.

Entre los factores que afectan a la detección de los patrones podemos destacar su tamaño, complejidad, orientación relativa a la cámara y condiciones de iluminación de la escena (ver Figura 31.47).

El **tamaño físico de la marca** afecta directamente a la facilidad de detección; a mayor tamaño de marca, mayor distancia puede ser cubierta. Por ejemplo, marcas de 7 cm de lado pueden ser detectadas hasta una distancia máxima de 40 cm (a una resolución de 640x480 píxeles). Si aumentamos el tamaño de la marca a 18cm, ésta será detectada hasta una distancia de 125cm.

Este rango de detección se ve también afectado por la **complejidad de la marca**. Los patrones simples (con grandes áreas de color blanco o negro) son detectados mejor.

La **orientación relativa** de la marca con respecto a la cámara afecta a la calidad de la detección. A mayor perpendicularidad entre el eje Z de la marca y el vector *look* de la cámara, la detección será peor.

Finalmente las **condiciones de iluminación** afectan enormemente a la detección de las marcas. El uso de materiales que no ofrezcan brillo especular, y que disminuyan el reflejo en las marcas mejoran notablemente la detección de las marcas.

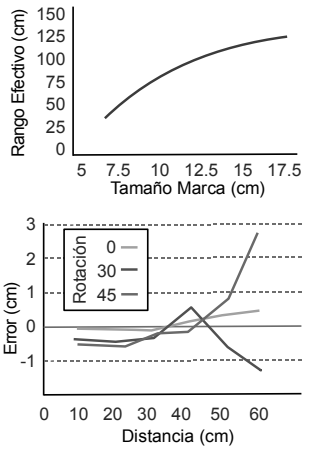


Figura 31.47: Arriba Relación entre el tamaño de la marca y el rango máximo de detección efectiva. **Abajo** Relación entre el error cometido, la distancia de detección, y el ángulo relativo entre la marca y la cámara.

31.20. Histórico de Percepciones

ARToolKit incorpora una función de tratamiento del histórico de percepciones para estabilizar el tracking. Este histórico se implementa en una función alternativa a `arGetTransMat` que, en realidad, utiliza únicamente la percepción anterior, llamada `arGetTransMatCont`. Mediante el uso de esta función se elimina gran parte del efecto de registro *tembloroso* (ver Figura 31.48). En esta figura se muestran diferentes valores de la componente de la posición en X en 40 frames de captura. Lo interesante de esta figura no son los valores de estas posiciones (intencionadamente distintos), sino la *forma* de la trayectoria. Empleando el histórico, la trayectoria resultante es mucho más suave.

En el listado de esta sección se muestra un ejemplo de utilización de esta función de histórico. La función de *callback* de teclado (líneas `8-16`) permite activar el uso del histórico (mediante la tecla *h*).

Histórico y precisión

La utilización del histórico suaviza la captura y consigue mayor estabilidad, aunque el resultado del tracking cuenta con menor precisión.

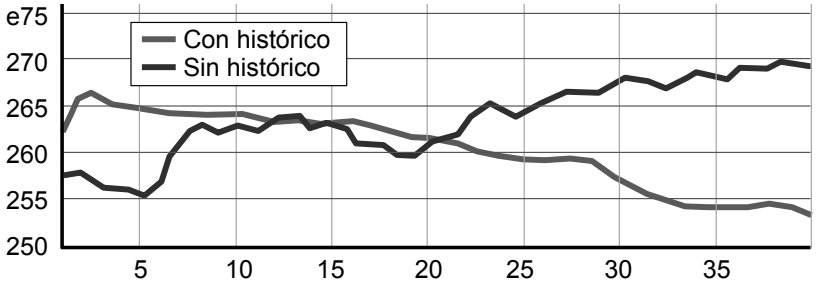


Figura 31.48: Comparativa de la trayectoria (en eje X) de 40 frames sosteniendo la marca manualmente activando el uso del histórico de percepciones. Se puede comprobar cómo la gráfica en el caso de activar el uso del histórico es mucho más suave.

Como el histórico requiere como parámetro la percepción anterior, no podrá utilizarse hasta que no dispongamos (al menos) de una percepción. Por esta razón, es necesario tener en cuenta este aspecto para llamar a la función de histórico `arGetTransMatCont` (cuando tengamos una percepción) o a la función sin histórico `arGetTransMat` la primera vez. Para esto, se utiliza otra variable llamada `useCont` (línea 4), que nos indica si ya podemos utilizar la función con histórico o no. Esta variable de comportamiento booleano se tendrá que poner a *false* (valor 0) cuando no se detecte la marca (línea 65).

Listado 31.25: Uso del histórico de percepciones.

```
// ==== Definicion de constantes y variables globales =============
int    patt_id;             // Identificador unico de la marca
double patt_trans[3][4];    // Matriz de transformacion de la marca
int    useCont = 0;         // Inicialmente no puede usar historico!
int    contAct = 0;         // Indica si queremos usar el historico

// ======== keyboard ============================================
static void keyboard(unsigned char key, int x, int y) {
  switch (key) {
  case 'H': case 'h':
    if (contAct) {contAct = 0; printf("Historico Desactivado\n");}
    else {contAct = 1; printf("Historico Activado\n");} break;
  case 0x1B: case 'Q': case 'q':
    cleanup(); exit(1); break;
  }
}

// ======== mainLoop ============================================
static void mainLoop(void) {
  ARUint8 *dataPtr;
  ARMarkerInfo *marker_info;
  int marker_num, j, k;

  double p_width     = 120.0;         // Ancho del patron (marca)
  double p_center[2] = {0.0, 0.0};    // Centro del patron (marca)

  // Capturamos un frame de la camara de video
  if((dataPtr = (ARUint8 *)arVideoGetImage()) == NULL) {
    // Si devuelve NULL es porque no hay un nuevo frame listo
    arUtilSleep(2);  return;  // Dormimos el hilo 2ms y salimos
  }

  argDrawMode2D();
  argDispImage(dataPtr, 0,0);    // Dibujamos lo que ve la camara

  // Detectamos la marca en el frame capturado (return -1 si error)
  if(arDetectMarker(dataPtr,100,&marker_info,&marker_num) < 0) {
    cleanup(); exit(0);   // Si devolvio -1, salimos del programa!
  }

  arVideoCapNext();      // Frame pintado y analizado...

  // Vemos donde detecta el patron con mayor fiabilidad
  for(j = 0, k = -1; j < marker_num; j++) {
    if(patt_id == marker_info[j].id) {
      if (k == -1) k = j;
      else if(marker_info[k].cf < marker_info[j].cf) k = j;
    }
  }

  if(k != -1) {   // Si ha detectado el patron en algun sitio...
    // Transformacion relativa entre marca y la camara real
    if (useCont && contAct) {
      arGetTransMatCont(&marker_info[k], patt_trans, p_center,
            p_width, patt_trans);
      printf ("Usando historico!!!\n");
    }
    else {
      useCont = 1;  // En la siguiente iteracion lo podemos usar!
      arGetTransMat(&marker_info[k],p_center,p_width,patt_trans);
      printf ("Sin historico...\n");
    }
    draw();       // Dibujamos los objetos de la escena
  } else {
    useCont = 0; printf ("Reset Historico (fallo deteccion)\n");
  }

  argSwapBuffers(); // Cambiamos el buffer con lo dibujado
}
```

Si el usuario activó el uso del histórico (y ya tenemos al menos una percepción previa de la marca, línea 53), utilizamos la función con histórico 54. En otro caso, llamaremos a la función de detección sin histórico 60 y activaremos su uso para la siguiente llamada 59.

31.21. Utilización de varios patrones

En muchas ocasiones querremos trabajar con más de un patrón. Un ejemplo típico es asociar cada patrón con un objeto 3D, o utilizarlos (como veremos en la siguiente sección) como mecanismos de interacción para realizar otras operaciones.

ARToolKit no proporciona ningún mecanismo directo para trabajar con varios patrones, por lo que es responsabilidad del programador mantener alguna estructura para guardar las matrices asociadas a cada marca detectada. A continuación veremos un sencillo ejemplo que define un tipo de datos propio `TObject` para guardar este tipo de información.

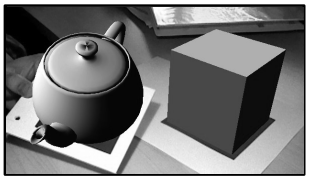


Figura 31.49: Asociación de diferentes modelos a varios patrones.

Listado 31.26: Utilización de varios patrones.

```
1  // ==== Definicion de estructuras =================================
2  struct TObject{
3    int id;                       // Identificador del patron
4    int visible;                  // Es visible el objeto?
5    double width;                 // Ancho del patron
6    double center[2];             // Centro del patron
7    double patt_trans[3][4];      // Matriz asociada al patron
8    void (* drawme)(void);        // Puntero a funcion drawme
9  };
10
11 struct TObject *objects = NULL;
12 int nobjects = 0;
13
14 // ==== addObject (Anade objeto a la lista de objetos) ============
15 void addObject(char *p, double w, double c[2], void(*drawme)(void))
16 {
17   int pattid;
18
19   if((pattid=arLoadPatt(p)) < 0) print_error ("Error en patron\n");
20   nobjects++;
21   objects = (struct TObject *)
22     realloc(objects, sizeof(struct TObject)*nobjects);
23
24   objects[nobjects-1].id = pattid;
25   objects[nobjects-1].width = w;
26   objects[nobjects-1].center[0] = c[0];
27   objects[nobjects-1].center[1] = c[1];
28   objects[nobjects-1].drawme = drawme;
29 }
30 // ==== draw****** (Dibujado especifico de cada objeto) ===========
31 void drawteapot(void) {
32   GLfloat material[]     = {0.0, 0.0, 1.0, 1.0};
33   glMaterialfv(GL_FRONT, GL_AMBIENT, material);
34   glTranslatef(0.0, 0.0, 60.0);
35   glRotatef(90.0, 1.0, 0.0, 0.0);
36   glutSolidTeapot(80.0);
37 }
38
39 void drawcube(void) {
40   GLfloat material[]     = {1.0, 0.0, 0.0, 1.0};
41   glMaterialfv(GL_FRONT, GL_AMBIENT, material);
42   glTranslatef(0.0, 0.0, 40.0);
43   glutSolidCube(80.0);
```

```
44 }
45
46 // ======== cleanup ================================================
47 static void cleanup(void) {   // Libera recursos al salir ...
48   arVideoCapStop();  arVideoClose();  argCleanup();
49   free(objects);  // Liberamos la memoria de la lista de objetos!
50   exit(0);
51 }
52
53 // ======== draw ====================================================
54 void draw( void ) {
55   double  gl_para[16];   // Esta matriz 4x4 es la usada por OpenGL
56   GLfloat light_position[]  = {100.0,-200.0,200.0,0.0};
57   int i;
58
59   argDrawMode3D();              // Cambiamos el contexto a 3D
60   argDraw3dCamera(0, 0);        // Y la vista de la camara a 3D
61   glClear(GL_DEPTH_BUFFER_BIT); // Limpiamos buffer de profundidad
62   glEnable(GL_DEPTH_TEST);
63   glDepthFunc(GL_LEQUAL);
64
65   for (i=0; i<nobjects; i++) {
66     if (objects[i].visible) {   // Si el objeto es visible
67       argConvGlpara(objects[i].patt_trans, gl_para);
68       glMatrixMode(GL_MODELVIEW);
69       glLoadMatrixd(gl_para);   // Cargamos su matriz de transf.
70
71       // La parte de iluminacion podria ser especifica de cada
72       // objeto, aunque en este ejemplo es general para todos.
73       glEnable(GL_LIGHTING);  glEnable(GL_LIGHT0);
74       glLightfv(GL_LIGHT0, GL_POSITION, light_position);
75       objects[i].drawme();      // Llamamos a su funcion de dibujar
76     }
77   }
78   glDisable(GL_DEPTH_TEST);
79 }
80
81 // ======== init ====================================================
82 static void init( void ) {
83   ARParam  wparam, cparam;   // Parametros intrinsecos de la camara
84   int xsize, ysize;          // Tamano del video de camara (pixels)
85   double c[2] = {0.0, 0.0};  // Centro de patron (por defecto)
86
87   // Abrimos dispositivo de video
88   if(arVideoOpen("") < 0) exit(0);
89   if(arVideoInqSize(&xsize, &ysize) < 0) exit(0);
90
91   // Cargamos los parametros intrinsecos de la camara
92   if(arParamLoad("data/camera_para.dat", 1, &wparam) < 0)
93     print_error ("Error en carga de parametros de camara\n");
94
95   arParamChangeSize(&wparam, xsize, ysize, &cparam);
96   arInitCparam(&cparam);   // Inicializamos la camara con çparam"
97
98   // Inicializamos la lista de objetos
99   addObject("data/simple.patt", 120.0, c, drawteapot);
100  addObject("data/identic.patt", 90.0, c, drawcube);
101
102  argInit(&cparam, 1.0, 0, 0, 0, 0);   // Abrimos la ventana
103 }
104
105 // ======== mainLoop ================================================
106 static void mainLoop(void) {
107   ARUint8 *dataPtr;
108   ARMarkerInfo *marker_info;
109   int marker_num, i, j, k;
110
111   // Capturamos un frame de la camara de video
112   if((dataPtr = (ARUint8 *)arVideoGetImage()) == NULL) {
113     // Si devuelve NULL es porque no hay un nuevo frame listo
114     arUtilSleep(2);  return;  // Dormimos el hilo 2ms y salimos
```

```
115   }
116
117   argDrawMode2D();
118   argDispImage(dataPtr, 0,0);    // Dibujamos lo que ve la camara
119
120   // Detectamos la marca en el frame capturado (ret -1 si error)
121   if(arDetectMarker(dataPtr, 100, &marker_info, &marker_num) < 0){
122     cleanup(); exit(0);   // Si devolvio -1, salimos del programa!
123   }
124
125   arVideoCapNext();      // Frame pintado y analizado...
126
127   // Vemos donde detecta el patron con mayor fiabilidad
128   for (i=0; i<nobjects; i++) {
129     for(j = 0, k = -1; j < marker_num; j++) {
130       if(objects[i].id == marker_info[j].id) {
131         if (k == -1) k = j;
132         else if(marker_info[k].cf < marker_info[j].cf) k = j;
133       }
134     }
135
136     if(k != -1) {   // Si ha detectado el patron en algun sitio...
137       objects[i].visible = 1;
138       arGetTransMat(&marker_info[k], objects[i].center,
139             objects[i].width, objects[i].patt_trans);
140     } else { objects[i].visible = 0; }  // El objeto no es visible
141   }
142
143   draw();           // Dibujamos los objetos de la escena
144   argSwapBuffers(); // Cambiamos el buffer
145 }
```

La estructura de datos base del ejemplo es `TObject`, definida en las líneas 2-9. En esta estructura, el último campo `drawme` es un puntero a función, que deberá ser asignado a alguna función que no reciba ni devuelva argumentos, y que se encargará de dibujar el objeto.

Mantendremos en `objects` (línea 11) una lista de objetos reconocibles en la escena. La memoria para cada objeto de la lista se reservará en la función `addObject` (líneas 15-29) mediante una llamada a `realloc` en 21-22. Esta memoria será liberada cuando finalice el programa en `cleanup` (línea 49).

De este modo, en la función de inicialización `init`, se llama a nuestra función `addObject` para cada objeto que deba ser reconocido, pasándole la ruta al fichero del patrón, el ancho, centro y el nombre de la función de dibujado asociada a esa marca (líneas 99-100). En este ejemplo se han definido dos sencillas funciones de dibujado de una tetera 31-37 y un cubo 39-44. En el caso de que una marca no tenga asociado el dibujado de un objeto (como veremos en los ejemplos de la sección 31.22), simplemente podemos pasar `NULL`.

En el bucle principal se han incluido pocos cambios respecto de los ejemplos anteriores. Únicamente en las líneas de código donde se realiza la comprobación de las marcas detectadas 129-142, se utiliza la lista de objetos global y se realiza la comparación con cada marca. Así, el bucle `for` externo se encarga de recorrer todos los objetos que pueden ser reconocidos por el programa 129, y se compara el factor de confianza de cada objeto con el que está siendo estudiado 133. A continuación se obtiene la matriz de transformación asociada al objeto en 139-140.

La función de dibujado (llamada en 144 y definida en 54-79) utiliza igualmente la información de la lista de objetos. Para cada objeto visible 65-66 se carga su matriz de transformación 67-69 y se llama a su función propia de dibujado 75.

Orientación a Objetos. Estos ejemplos están realizados en C intentando crear ejemplos de código mínimos. En la integración con Ogre sería conveniente crear objetos para el tracking y despliegue con la información necesaria.

31.22. Relación entre Coordenadas

Como hemos visto en los ejemplos de los capítulos anteriores, ARToolKit calcula la posición de la marca en coordenadas de la cámara.

Puede entenderse como un posicionamiento de la cámara en una posición del espacio (correspondiente a la distancia y rotación relativa entre la marca y la cámara real), conservando el origen del mundo en el centro de la marca. Esta *idea* es únicamente una ayuda para entender el posicionamiento real, ya que internamente es una transformación que se realiza en la pila *ModelView*, siendo equivalente mover la cámara y mantener el objeto (la marca) estática o a la inversa.

Coordenadas Marca

Como vimos en el capítulo de OpenGL del módulo 2, el sistema de coordenadas de la marca sigue el mismo convenio que OpenGL (ver Figura 31.50), por lo que si utilizamos la idea intuitiva de que el origen del SRU está en la marca, podemos posicionar fácilmente los objetos con OpenGL, ya que las transformaciones se aplican siguiendo la misma notación.

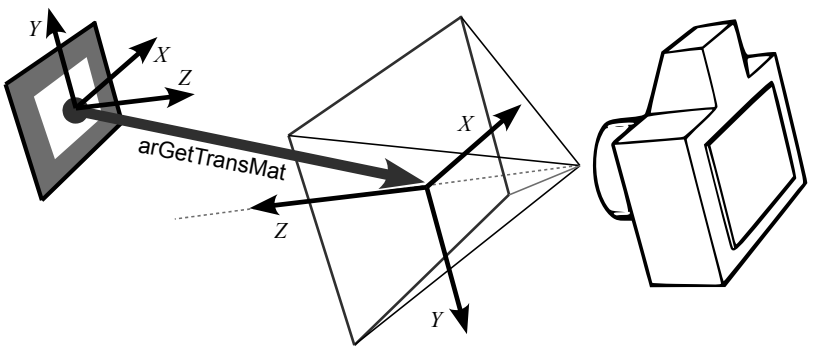


Figura 31.50: Relación entre el sistema de coordendas de la marca y el sistema de coordenadas de la cámara.

De este modo, como se representa en la Figura 31.50, la llamada a `arGetTransMat` nos posiciona la cámara en relación a la marca. Es como si aplicáramos la transformación marcada por la flecha de color rojo, con origen del sistema de referencia en la marca.

De hecho, si en el ejemplo del *Hola Mundo* imprimimos la columna de más a la derecha de la matriz de transformación de la marca, podemos comprobar cómo si movemos la marca hacia arriba (en dirección a su eje Y), el valor de posición en ese Y decrecerá (debido a que el sentido está invertido respecto de la cámara). De forma análoga ocurrirá con el eje Z. Por el contrario, si desplazamos la marca hacia la derecha, el eje X de la matriz crecerá, debido a que ambos sistemas de coordenadas emplean el mismo convenio (ver Figura 31.50).

Continuaremos trabajando con la idea mucho más intuitiva de considerar que la marca es estática y que lo que se desplaza es la cámara (aunque, como veremos, la marca puede en realidad desplazarse y la idea de considerar que la cámara es la móvil seguirá siendo últil).

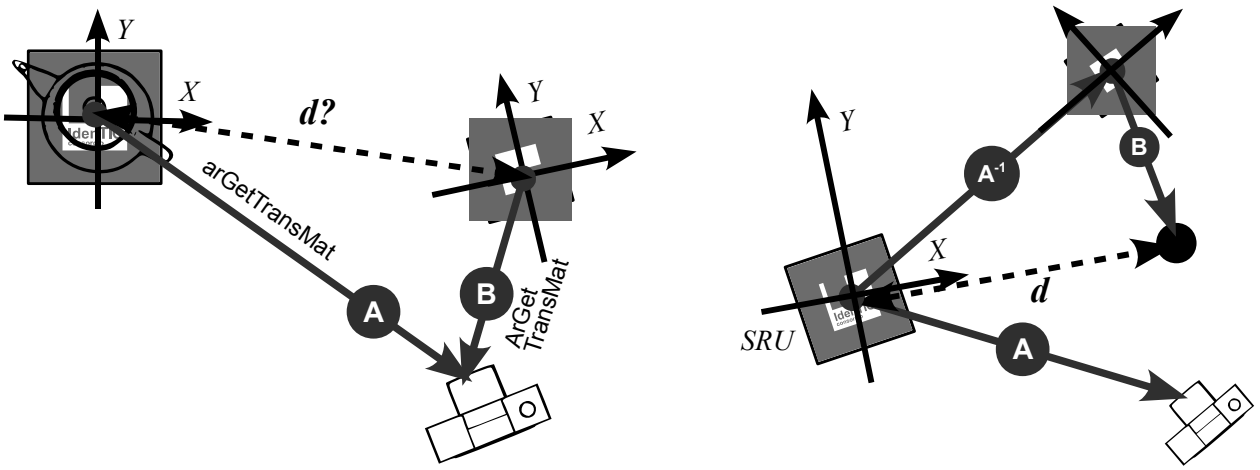


Figura 31.53: Utilización de la transformación inversa a la marca de Identic A^{-1} para calcular la distancia entre marcas. Primero *cargamos* la matriz asociada a la marca de marca de Identic, (estableciendo implícitamente el origen del SRU ahí) y de ahí aplicamos la transformación inversa A^{-1}. A continuación aplicamos la matriz B asociada a la segunda marca. La distancia al origen del SRU es la distancia entre marcas.

Veamos a continuación un ejemplo que trabajará con las relaciones entre la cámara y la marca. Tener claras estas transformaciones nos permitirá cambiar entre sistemas de coordenadas y trabajar con las relaciones espaciales entre las marcas que no son más que transformaciones entre diferentes sistemas de coordenadas. En este programa queremos variar la intensidad de color rojo de la tetera según la distancia de una marca auxiliar (ver Figura 31.51).

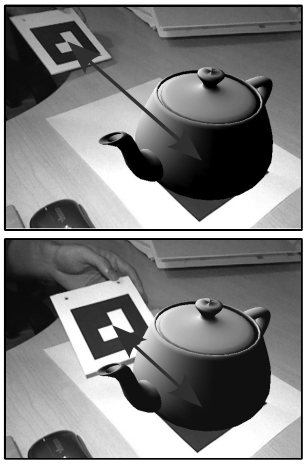


Figura 31.51: Salida del ejemplo de cálculo de distancias entre marcas. La intensidad de la componente roja del color varía en función de la distancia entre marcas.

El problema nos pide calcular la distancia entre dos marcas. ARToolKit, mediante la llamada a la función `arGetTransMat` nos devuelve la transformación relativa entre la marca y la cámara. Para cada marca podemos obtener la matriz de transformación *relativa* hacia la cámara. Podemos verlo como la transformación que nos posiciona la cámara en el lugar adecuado para que, dibujando los objetos en el origen del SRU, se muestren de forma correcta. ¿Cómo calculamos entonces la distancia entre ambas marcas?

En el diagrama de la Figura 31.53 se resume el proceso. Con la llamada a `arGetTransMat` obtenemos una transformación relativa al sistema de coordenadas de cada cámara (que, en el caso de una única marca, podemos verlo como si fuera el origen del *SRU* donde dibujaremos los objetos). En este caso, tenemos las flechas que parten de la marca y posicionan relativamente la cámara señaladas con A y B.

Podemos calcular la transformación entre marcas empleando la inversa de una transformación, que equivaldría a *viajar* en sentido contrario. Podemos imaginar realizar la transformación contraria; partiendo del origen del *SRU*, viajamos hasta la marca de Identic (aplicando A^{-1}, y de ahí aplicamos la transformación B (la que nos posicionaría desde la segunda marca hasta la cámara). Así, llegamos al punto final (señalado con el círculo de color negro en la Figura 31.53.derecha). Su distancia al origen del *SRU* será la distancia entre marcas.

La codificación de esta operación es directa. En la línea [33] del listado anterior, si ambos objetos son visibles [32] se calcula la inversa de la transformación a la marca de Identic (que está en la posición 0 de la lista de objetos). Esta nueva matriz `m` se multiplica a continuación con la matriz de transformación de la segunda marca [34].

Listado 31.27: Ejemplo de cálculo de distancias entre marcas.

```
1  // ======== drawteapot ============================================
2  void drawteapot(void) {
3    GLfloat material[]     = {0.0, 0.0, 0.0, 1.0};
4    float value = 0.0;            // Intensidad del gris a dibujar
5
6    // La intensidad se asigna en funcion de la distancia "dist01"
7    // dist01 es la distancia entre objeto 1 y 2 (global)
8    // Mapear valor intensidad linealmente entre 160 y 320 ->(1..0)
9    value = (320 - dist01) / 160.0;
10   if (value < 0) value = 0;  if (value > 1) value = 1;
11   material[0] = value;
12
13   glMaterialfv(GL_FRONT, GL_AMBIENT, material);
14   glTranslatef(0.0, 0.0, 60.0);
15   glRotatef(90.0, 1.0, 0.0, 0.0);
16   glutSolidTeapot(80.0);
17 }
18
19 // ======== draw ================================================
20 void draw( void ) {
21   double  gl_para[16];   // Esta matriz 4x4 es la usada por OpenGL
22   GLfloat light_position[]  = {100.0,-200.0,200.0,0.0};
23   double m[3][4], m2[3][4];
24   int i;
25
26   argDrawMode3D();              // Cambiamos el contexto a 3D
27   argDraw3dCamera(0, 0);        // Y la vista de la camara a 3D
28   glClear(GL_DEPTH_BUFFER_BIT); // Limpiamos buffer de profundidad
29   glEnable(GL_DEPTH_TEST);
30   glDepthFunc(GL_LEQUAL);
31
32   if (objects[0].visible && objects[1].visible) {
33     arUtilMatInv(objects[0].patt_trans, m);
34     arUtilMatMul(m, objects[1].patt_trans, m2);
35     dist01 = sqrt(pow(m2[0][3],2)+pow(m2[1][3],2)+pow(m2[2][3],2));
36     printf ("Distancia objects[0] y objects[1]= %G\n", dist01);
37   }
38
39   for (i=0; i<nobjects; i++) {
40     if ((objects[i].visible) && (objects[i].drawme != NULL)) {
41       argConvGlpara(objects[i].patt_trans, gl_para);
42       glMatrixMode(GL_MODELVIEW);
43       glLoadMatrixd(gl_para);   // Cargamos su matriz de transf.
44
45       glEnable(GL_LIGHTING);  glEnable(GL_LIGHT0);
46       glLightfv(GL_LIGHT0, GL_POSITION, light_position);
47       objects[i].drawme();      // Llamamos a su funcion de dibujar
48     }
49   }
50   glDisable(GL_DEPTH_TEST);
51 }
52
53 // ======== init ================================================
54 static void init( void ) {
55   // La parte inicial de init es igual que en ejemplos anteriores
56
57   // Inicializamos la lista de objetos
58   addObject("data/identic.patt", 120.0, c, drawteapot);
59   addObject("data/simple.patt", 90.0, c, NULL);
60
61   argInit(&cparam, 1.0, 0, 0, 0, 0);   // Abrimos la ventana
62 }
```

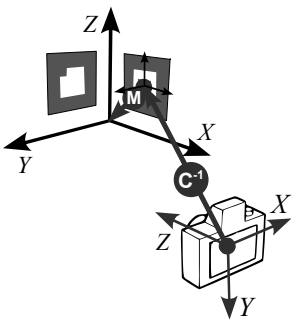


Figura 31.52: Esquema de posicionamiento global utilizando transformaciones inversas.

Posicionamiento Global

El uso de transformaciones inversas nos permiten obtener coordenadas de posicionamiento global, como se muestra en la Figura 31.52. En ese ejemplo, bastará con conocer la matriz de transformación entre la marca y el origen del SRU (expresada como M), para obtener el posicionamiento global de la cámara respecto del SRU, multiplicando la inversa de la transformación relativa de la marca C^{-1} con la matriz M.

Recordemos que la última columna de las matrices netas de transformación codifican la traslación. De este modo, basta con calcular el módulo del vector que une el punto trasladado con el origen para calcular la distancia, que se guarda en la variable global `dist01` 35 (distancia entre los objetos 0 y 1). Para cambiar el color de la tetera en función de esta distancia usamos una función lineal que asigna el nivel de rojo (entre 0 y 1) según la distancia en las líneas 9-11 (si $d \leq 16cm, r = 1$, si $d > 32cm, r = 0$, y linealmente los intermedios).

31.23. Integración con OpenCV y Ogre

En esta sección estudiaremos primero cómo realizar una captura de vídeo y despliegue en el fondo de una ventana de Ogre para, posteriormente, convertir el tipo de datos relativos a un *Frame* en OpenCV para ser utilizados en ARToolKit (ver Figura 31.54).

Figura 31.54: Ejemplo de integración que se describirá en esta sección.

La clase *VideoManager* se encarga de la gestión de las fuentes de vídeo. Podrían instanciarse tantos objetos de esta clase como fueran necesarios, pero habría que modificar la implementación de la llamada a *createBackground* y a *DrawCurrentFrame*. El *VideoManager* abstrae a la aplicación de Ogre del uso de OpenCV. Únicamente el *VideoManager* conoce los dispositivos de captura (línea 10 del archivo de cabecera). El puntero al *sceneManager* de Ogre es necesario como veremos más adelante para la creación y actualización del plano que utilizaremos para desplegar la salida del dispositivo de vídeo en la escena.

El *VideoManager* trabaja internamente con dos tipos de datos para la representación de los *frames* (ver líneas 11-12), que pueden ser convertidos entre sí fácilmente. El IplImage ha sido descrito anteriormente en la sección. cv::Mat es una extensión para trabajar con matrices de cualquier tamaño y tipo en C++, que permiten acceder cómodamente a los datos de la imagen. De este modo, cuando se capture un frame, el *VideoManager* actualizará los punteros anteriores de manera inmediata.

Listado 31.28: VideoManager.h

```
1  #include <Ogre.h>
2  #include <iostream>
3  #include "cv.h"
4  #include "highgui.h"
5
6  class VideoManager {
7   private:
8    void createBackground(int cols, int rows);
9    void ReleaseCapture();
10   CvCapture* _capture;
11   IplImage* _frameIpl;
12   cv::Mat* _frameMat;
13   Ogre::SceneManager* _sceneManager;
14
15  public:
16   VideoManager(int device, int w, int h,
17                Ogre::SceneManager* sm);
18   ~VideoManager();
19   void UpdateFrame();
20   IplImage* getCurrentFrameIpl();
21   cv::Mat* getCurrentFrameMat();
22   void DrawCurrentFrame();
23 };
```

El constructor de la clase (líneas 4-8 del siguiente listado) se encarga de obtener el dispositivo de captura e inicializarlo con las dimensiones especificadas como parámetro al constructor. Esto es necesario, ya que las cámaras generalmente permiten trabajar con varias resoluciones. En el destructor se libera el dispositivo de captura creado por OpenCV (línea 12).

Listado 31.29: VideoManager.cpp

```
1  #include "VideoManager.h"
2  VideoManager::VideoManager(int device, int w, int h,
3                 Ogre::SceneManager* sm){
4    _sceneManager = sm;
5    _capture = cvCreateCameraCapture(device);
6    cvSetCaptureProperty(_capture, CV_CAP_PROP_FRAME_WIDTH, w);
7    cvSetCaptureProperty(_capture, CV_CAP_PROP_FRAME_HEIGHT, h);
8    createBackground (w, h); _frameIpl = NULL; _frameMat = NULL;
9  }
10
11 VideoManager::~VideoManager(){
12   cvReleaseCapture(&_capture); delete _frameIpl; delete _frameMat;
13 }
14 // ================================================================
15 // createBackground: Crea el plano sobre el que dibuja el video
16 void VideoManager::createBackground(int cols, int rows){
17   Ogre::TexturePtr texture=Ogre::TextureManager::getSingleton().
18     createManual("BackgroundTex", // Nombre de la textura
19     Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
20     Ogre::TEX_TYPE_2D,   // Tipo de la textura
21     cols, rows, 0,       // Filas, columas y Numero de Mipmaps
22     Ogre::PF_BYTE_BGRA,
23     Ogre::HardwareBuffer::HBU_DYNAMIC_WRITE_ONLY_DISCARDABLE);
24
```

```
25    Ogre::MaterialPtr mat = Ogre::MaterialManager::getSingleton().
26      create("Background",
27            Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
28    mat->getTechnique(0)->getPass(0)->createTextureUnitState();
29    mat->getTechnique(0)->getPass(0)->setDepthCheckEnabled(false);
30    mat->getTechnique(0)->getPass(0)->setDepthWriteEnabled(false);
31    mat->getTechnique(0)->getPass(0)->setLightingEnabled(false);
32    mat->getTechnique(0)->getPass(0)->getTextureUnitState(0)->
33      setTextureName("BackgroundTex");
34
35    // Creamos un rectangulo que cubra toda la pantalla
36    Ogre::Rectangle2D* rect = new Ogre::Rectangle2D(true);
37    rect->setCorners(-1.0, 1.0, 1.0, -1.0);
38    rect->setMaterial("Background");
39
40    // Dibujamos el background antes que nada
41    rect->setRenderQueueGroup(Ogre::RENDER_QUEUE_BACKGROUND);
42
43    Ogre::SceneNode* node = _sceneManager->getRootSceneNode()->
44      createChildSceneNode("BackgroundNode");
45    node->attachObject(rect);
46  }
47  // ================================================================
48  // UpdateFrame: Actualiza los punteros de frame Ipl y frame Mat
49  void VideoManager::UpdateFrame(){
50    _frameIpl = cvQueryFrame(_capture);
51    _frameMat = new cv::Mat(_frameIpl);
52  }
53  // = IplImage* getCurrentFrameIpl ================================
54  IplImage* VideoManager::getCurrentFrameIpl(){ return _frameIpl; }
55  // = IplImage* getCurrentFrameMat ================================
56  cv::Mat* VideoManager::getCurrentFrameMat(){  return _frameMat; }
57  // ================================================================
58  // DrawCurrentFrame: Despliega el ultimo frame actualizado
59  void VideoManager::DrawCurrentFrame(){
60    if(_frameMat->rows==0) return;
61    Ogre::TexturePtr tex = Ogre::TextureManager::getSingleton().
62       getByName("BackgroundTex",
63       Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
64    Ogre::HardwarePixelBufferSharedPtr pBuffer = tex->getBuffer();
65
66    pBuffer->lock(Ogre::HardwareBuffer::HBL_DISCARD);
67    const Ogre::PixelBox& pixelBox = pBuffer->getCurrentLock();
68
69    Ogre::uint8* pDest = static_cast<Ogre::uint8*>(pixelBox.data);
70    for(int j=0;j<_frameMat->rows;j++) {
71      for(int i=0;i<_frameMat->cols;i++) {
72        int idx = ((j) * pixelBox.rowPitch + i )*4;
73        pDest[idx] = _frameMat->data[(j*_frameMat->cols+i)*3];
74        pDest[idx+1] = _frameMat->data[(j*_frameMat->cols+i)*3+1];
75        pDest[idx+2] = _frameMat->data[(j*_frameMat->cols+i)*3+2];
76        pDest[idx+3] = 255;
77      }
78    }
79    pBuffer->unlock();
80    Ogre::Rectangle2D* rect = static_cast<Ogre::Rectangle2D*>
81      (_sceneManager->getSceneNode("BackgroundNode")->
82          getAttachedObject(0));
83  }
```

El método *createBackground* se encarga de crear un plano sobre el que se actualizará el vídeo obtenido de la cámara. Además de las propiedades relativas a desactivar el uso del *depthbuffer* o la iluminación (líneas 29-31), es importante indicar que el buffer asociado a ese material va a ser actualizado muy frecuentemente, mediante la constante de uso del buffer definida en la línea 23). Posteriormente crearemos un

rectángulo 2D (coordenadas paramétricas en la línea 37), que añadiremos al grupo que se dibujará primero (línea 41). Finalmente se añade el nodo, con el plano y la textura, a la escena (líneas 43-45) empleando el puntero al *SceneManager* indicado en el constructor.

La actualización de la textura asociada al plano se realiza en el método *DrawCurrentFrame*, que se encarga de acceder a nivel de píxel y copiar el valor de la imagen obtenida por la webcam. Para ello se obtiene un puntero compartido al *pixel buffer* (línea 64), y obtiene el uso de la memoria con exclusión mutua (línea 67). Esa región será posteriormente liberada en la línea 79. La actualización de la región se realiza recorriendo el frame en los bucles de las líneas 70-77, y especificando el color de cada píxel en valores RGBA (formato de orden de componentes configurado en línea 22).

El frame obtenido será utilizado igualmente por ARToolKit para detectar las marcas y proporcionar la transformación relativa. Igualmente se ha definido una clase para abstraer del uso de ARToolKit a la aplicación de Ogre. El archivo de cabecera del *ARTKDetector* se muestra a continuación.

Listado 31.30: ARTKDetector.h

```
1  #include <AR/ar.h>
2  #include <AR/gsub.h>
3  #include <AR/param.h>
4  #include <Ogre.h>
5  #include <iostream>
6  #include "cv.h"
7
8  class ARTKDetector {
9   private:
10   ARMarkerInfo *_markerInfo;
11   int _markerNum;
12   int _thres;
13   int _pattId;
14   int _width; int _height;
15   double _pattTrans[3][4];
16   bool _detected;
17
18   int readConfiguration();
19   int readPatterns();
20   void Gl2Mat(double *gl, Ogre::Matrix4 &mat);
21
22  public:
23   ARTKDetector(int w, int h, int thres);
24   ~ARTKDetector();
25   bool detectMark(cv::Mat* frame);
26   void getPosRot(Ogre::Vector3 &pos, Ogre::Vector3 &look,
27          Ogre::Vector3 &up);
28 };
```

En este sencillo ejemplo únicamente se han creado dos clases para la detección de la marca *detectMark*, y para obtener los vectores de posicionamiento de la cámara en función de la última marca detectada (*getPosRot*).

Complétame!

Sería conveniente añadir nuevos métodos a la clase ARTKDetector, que utilice una clase que encapsule los marcadores con su ancho, alto, identificador de patrón, etc...

Listado 31.31: ARTKDetector.cpp

```
1  #include "ARTKDetector.h"
2
3  ARTKDetector::ARTKDetector(int w, int h, int thres){
4    _markerNum=0;   _markerInfo=NULL;  _thres = thres;
5    _width = w;     _height = h;       _detected = false;
6    readConfiguration();
7    readPatterns();
8  }
9  ARTKDetector::~ARTKDetector(){  argCleanup();  }
10 // ================================================================
```

```
11 // readPatterns: Carga la definicion de patrones
12 int ARTKDetector::readPatterns(){
13   if((_pattId=arLoadPatt("data/simple.patt")) < 0) return -1;
14   return 0;
15 }
16 // ============================================================
17 // readConfiguration: Carga archivos de configuracion de camara...
18 int ARTKDetector::readConfiguration(){
19   ARParam  wparam, cparam;
20   // Cargamos los parametros intrinsecos de la camara
21   if(arParamLoad("data/camera_para.dat",1, &wparam) < 0) return -1;
22
23   arParamChangeSize(&wparam, _width, _height, &cparam);
24   arInitCparam(&cparam);   // Inicializamos la camara con çparam"
25   return 0;
26 }
27 // ============================================================
28 // detectMark (FIXME): Ojo solo soporta una marca de tamano fijo!
29 bool ARTKDetector::detectMark(cv::Mat* frame) {
30   int j, k;
31   double p_width     = 120.0;        // Ancho de marca... FIJO!
32   double p_center[2] = {0.0, 0.0};   // Centro de marca.. FIJO!
33
34   _detected = false;
35   if(frame->rows==0) return _detected;
36   if(arDetectMarker(frame->data, _thres,
37                     &_markerInfo, &_markerNum) < 0){
38     return _detected;
39   }
40   for(j=0, k=-1; j < _markerNum; j++) {
41     if(_pattId == _markerInfo[j].id) {
42       if (k == -1) k = j;
43       else if(_markerInfo[k].cf < _markerInfo[j].cf) k = j;
44     }
45   }
46   if(k != -1) {   // Si ha detectado el patron en algun sitio...
47     arGetTransMat(&_markerInfo[k], p_center, p_width, _pattTrans);
48     _detected = true;
49   }
50   return _detected;
51 }
52 // ============================================================
53 // Gl2Mat: Utilidad para convertir entre matriz OpenGL y Matrix4
54 void ARTKDetector::Gl2Mat(double *gl, Ogre::Matrix4 &mat){
55   for(int i=0;i<4;i++) for(int j=0;j<4;j++) mat[i][j]=gl[i*4+j];
56 }
57 // ============================================================
58 // getPosRot: Obtiene posicion y rotacion para la camara
59 void ARTKDetector::getPosRot(Ogre::Vector3 &pos,
60                              Ogre::Vector3 &look, Ogre::Vector3 &up){
61   if (!_detected) return;
62
63   double glAuxd[16]; Ogre::Matrix4 m;
64   argConvGlpara(_pattTrans,glAuxd);
65   Gl2Mat(glAuxd, m);   // Convertir a Matrix4 de Ogre
66
67   m[0][1]*=-1; m[1][1]*=-1;  m[2][1]*=-1; m[3][1]*=-1;
68   m = m.inverse();
69   m = m.concatenate(Ogre::Matrix4(
70       Ogre::Quaternion(Ogre::Degree(90), Ogre::Vector3::UNIT_X)));
71   pos  = Ogre::Vector3 (m[3][0],         m[3][1],        m[3][2]);
72   look = Ogre::Vector3 (m[2][0]+m[3][0], m[2][1]+m[3][1],
73                                                  m[2][2]+m[3][2]);
74   up   = Ogre::Vector3 (m[1][0],         m[1][1],        m[1][2]);
75 }
```

Esta clase está preparada para trabajar con una única marca (cargada en el método privado *readPatterns*), con sus variables asociadas definidas como miembros de la clase ARTKDetector (líneas 11-16). El siguiente listado implementa el *ARTKDetector*.

El método *getPosRot* es el único que vamos a comentar (el resto son una traducción directa de la funcionalidad estudiada anteriormente). En la línea [65] se utiliza un método de la clase para transformar la matriz de OpenGL a una *Matrix4* de Ogre. Ambas matrices son matrices columna!, por lo que habrá que tener cuidado a la hora de trabajar con ellas (el campo de traslación viene definido en la fila inferior, y no en la columna de la derecha como es el *convenio habitual*).

Las operaciones de la línea [67] sirven para invertir el eje Y de toda la matriz, para cambiar el criterio del sistema de ejes de *mano izquierda* a *mano derecha* (Ogre). Invertimos la matriz y aplicamos una rotación de 90 grados en X para tener el mismo sistema con el que hemos trabajado en sesiones anteriores del curso.

Finalmente calculamos los vectores de *pos*, *look* y *up* (líneas [71-74]) que utilizaremos en el bucle de dibujado de Ogre.

El uso de estas clases de utilidad es directa desde el FrameListener. Será necesario crear dos variables miembro del *VideoManager* y *ARTKDetector* (líneas [7-8]). En el método *frameStarted* bastará con llamar a la actualización del frame (líneas [15-16]). La detección de la marca se realiza pasándole el puntero de tipo Mat del *VideoManager* al método *detectMark* del *Detector*.

Listado 31.32: MyFrameListener.cpp

```
1  #include "MyFrameListener.h"
2
3  MyFrameListener::MyFrameListener(Ogre::RenderWindow* win,
4                 Ogre::Camera* cam, Ogre::SceneNode *node,
5                 Ogre::OverlayManager *om, Ogre::SceneManager *sm) {
6    // Omitido el resto del codigo...
7    _videoManager = new VideoManager(1, 640, 480, _sceneManager);
8    _arDetector = new ARTKDetector(640, 480, 100);
9  }
10
11 bool MyFrameListener::frameStarted(const Ogre::FrameEvent& evt) {
12   // Omitido el resto del codigo...
13   Ogre::Vector3 pos;  Ogre::Vector3 look;   Ogre::Vector3 up;
14
15   _videoManager->UpdateFrame();
16   _videoManager->DrawCurrentFrame();
17   if (_arDetector->detectMark(_videoManager->getCurrentFrameMat()))
       {
18     _arDetector->getPosRot(pos, look, up);
19     _camera->setPosition(pos);
20     _camera->lookAt(look);
21     _camera->setFixedYawAxis(true, up);
22   }
23 }
```

31.24. Consideraciones Finales

Como no podía ser de otra forma, se han quedado muchas cosas en el tintero relativas al uso de ARToolKit, como el uso de patrones multimarca, o el trabajo exhaustivo con cambios entre sistemas de coordenadas. Se recomienda como ejercicio ampliar la funcionalidad de la clase *ARTKDetector* para que soporte el uso del histórico de percepciones y la carga de múltiples marcas.

C++
BJARNE STROUSTRUP
ILUMINACIÓN
VALIDACIÓN Y PRUEBAS
NETWORKING

Anexos

NETWORKING
UMINACIÓN
BJARNE STROUSTRUP

Anexo A

Introducción a OgreFramework

Sergio Fernández Durán

Cada vez que creamos un nuevo proyecto con Ogre, existe un grupo de elementos que necesitamos instanciar para crear un primer prototipo. Este conjunto de clases y objetos se repiten con frecuencia para todos nuestros proyectos. En este capítulo estudiaremos los fundamentos de OgreFramework, que ofrece una serie de facilidades muy interesantes.

A.1. Introducción

OgreFramework fué creado por *Philip Allgaier* en 2009 y ofrece al programador una serie de herramientas para la gestión de Ogre como:

- Un sistema de Estado de Juego.
- Una interfaz gráfica de usuario.
- Diferentes modos de entrada.
- Carga de escenas.
- Manipulación manual de materiales.

El uso de este framework ayuda a **separar** la entidad del motor de renderizado de nuestro juego pudiendo combinarlo con nuevos módulos como: el motor de físicas o un middleware de comunicaciones. Estos factores pueden ayudar a gestionar mejor la coordinación entre varios motores, como la gestión del tiempo entre el motor de físicas y el motor de renderizado.

Existen dos ramas del framework para Ogre, *Basic* y *Advanced*. La diferencia entre los dos es la cantidad de recursos que manejan. *Basic OgreFramework* contiene los recursos **mínimos** necesarios para crear una instancia del motor Ogre y que muestre una escena como ejemplo. Y *Advanced OgreFramework* incluye un **conjunto completo** de herramientas y clases para una gestión avanzada de nuestra aplicación.

OgreFramework está desarrollado en *C++* sobre Ogre y es multiplataforma, sin embargo, es necesario realizar algunas modificaciones en el framework para que pueda funcionar en OSX. Una de las mayores ventajas de utilizar esta tecnología es la de poder automatizar la carga inicial del motor y poder configurar Ogre con llamadas simples a OgreFramework.

A.2. Basic OgreFramework

Una de las mejores formas de iniciarse con Ogre es mediante esta tecnología. Con esta herramienta no es necesario conocer la arquitectura del motor para poder crear un primer ejecutable y ayuda a conocer cuales son sus funciones básicas. Este paquete de herramientas incluye:

- Inicialización de Ogre.
- Bucle de renderizado personalizable.
- Escena básica.
- Gestión de teclado básica.
- Crear capturas de pantallas (Screenshots).
- Información básica de la escena (FPS, contador batch. . .).

Para conseguir la última versión de *Basic OgreFramework* clonaremos el proyecto con mercurial desde la siguiente dirección:

```
hg clone https://bitbucket.org/spacegaier/basicogreframework
```

Cuando descargamos el proyecto, podemos crear un ejecutable de OgreFramework directamente con los ficheros descargados. Una vez hemos compilado y ejecutado el código de ejemplo, nos encontraremos con la pantalla mostrada en la Figura A.1.

Su estructura es muy sencilla y explicaremos con más profundidad el funcionamiento del framework en la siguiente sección.

Figura A.1: Resultado del primer ejemplo con Ogre Framework.

A.2.1. Arquitectura

La rama *Basic* de OgreFramework contiene en exclusiva la instanciación del motor de renderizado. Se basa en el patrón *Singleton*. Cada vez que queramos hacer uso de alguna llamada al motor será necesario recuperar el puntero al Singleton del objeto. Dentro de la clase OgreFramework nos encontramos con el arranque de los siguientes gestores:

> OgreRoot, Camera, RenderWindow, ViewPort, SceneManager, Log, Timer, InputManager, etc.

Todos estos gestores de la clase OgreFramework serán explicados con más detalle en la siguiente sección, donde hablaremos de la rama '*Advanced*'. Para poder crear un ejemplo basado en Basic OgreFramework es necesario:

- Llamar a las funciones de inicio, configuración y ejecución del motor.
- Un puntero a una instancia de OgreFramework.
- Una entidad y un nodo escena.
- Una variable booleana que gestione el bucle principal del juego.

Ahora veremos los pasos necesarios para rellenar de contenido nuestra escena. Para ello crearemos nodos y escenas igual que las creamos en Ogre pero llamando al puntero de OgreFramework, como se muestra en el siguiente listado.

Listado A.1: Creación de escena.

```
1  void ExampleApp::createScene()
2  {
3    OgreFramework::getSingletonPtr()->
4      m_pSceneMgr->setSkyBox(true, "Examples/Sky");
5    OgreFramework::getSingletonPtr()->
6      m_pSceneMgr->createLight("Light")->setPosition(75,75,75);
7
8    m_pSinbadEntity = OgreFramework::getSingletonPtr()->
9      m_pSceneMgr->createEntity("SinbadEntity", "Sinbad.mesh");
10   m_pSinbadNode = OgreFramework::getSingletonPtr()->
11     m_pSceneMgr->getRootSceneNode()->createChildSceneNode("
           SinbadNode");
12   m_pSinbadNode->attachObject(m_pSinbadEntity);
13 }
```

Como podemos observar, para crear una entidad y un nodo, es necesario acceder al gestor de escena mediante el puntero *Singleton* a OgreFramework. Se realizará la misma acción para crear las luces de la escena y el Skybox. Esta restricción nos ayudará a mantener **separado** el motor Ogre de nuestro modelo y permitirá la integración con otros módulos de forma más homogénea para el desarrollador.

Realmente, cuando accedamos a uno de los gestores de OgreFramework como el *SceneManager*, utilizaremos las mismas primitivas que utilizabamos en Ogre. Por lo tanto, podremos usar la documentación estándar de Ogre.

Para gestionar nuestro bucle del juego necesitaremos la variable booleana *m_bShutdown* la cual nos permitirá cerrar la aplicación en cualquier momento. Si se produce un error inesperado, también saldrá del bucle principal.

En cada iteración del bucle:

- Se capturan las órdenes introducidas a través de teclado y ratón para mostrar nuevos eventos en la siguiente iteración.
- Se guarda el tiempo transcurrido desde el inicio.
- Se actualiza el motor Ogre con el tiempo del último frame generado.
- Se renderiza un frame.
- Se calcula el tiempo transcurrido desde el último frame generado.

A.3. Advanced OgreFramework

Para poder realizar un proyecto más complejo en Ogre, este framework nos sirve como **núcleo**. Este paquete de herramientas incluye lo siguiente:

- Inicialización de OGRE.
- Carga de recursos básicos.
- Gestión de dispositivos de entrada basado en OIS.
- Personalización del bucle de renderizado.
- Jerarquía basada en estados.
- Cargador de escenas mediante XML con DotSceneLoader.
- Interfaz gráfica de usuario basada en SdkTrays.
- Manipulación de materiales en código.

Para conseguir la última versión de Advanced OgreFramework clonaremos el proyecto con mercurial desde la siguiente dirección:

```
hg clone https://bitbucket.org/spacegaier/advancedogreframework
```

Una vez hayamos descargado el proyecto, podremos generar un ejecutable sin realizar ninguna modificación. Esta rama de OgreFramework nos proporciona tres estados distintos en la aplicación: Menu, Pausa y Juego. Algo básico para cualquier prototipo.

También nos proporciona un gestor de interfaz basado en SdkTrays (el mismo que utiliza Ogre para sus ejemplos) con el que podemos gestionar los estados del juego.

La intefaz incluye por defecto un conjunto de paneles *debug* que permiten conocer los FPS, el nombre del estado actual, la posición de la cámara, panel de ayuda, etc...

Podremos agregar y editar *Overlays* a los estados *Menú* para personalizar nuestra interfaz. Cada vez que entremos en un estado, se generarán los *widgets* que aparecen en pantalla y se crean los elementos de la interfaz específicos para ese estado. Cuando salimos o pausamos ese estado para entrar en uno nuevo, los elementos de la interfaz se destruyen y desaparecen de la escena. Podremos modificar los Widgets y elementos de la interfaz editando sus propiedades y materiales.

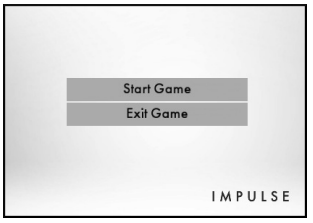


Figura A.2: Ejemplo básico de Widgets.

Dedicaremos un capítulo para explicar detalladamente cómo se gestiona la interfaz mediante SdkTrays y cómo podemos crear nuevos elementos. También explicaremos cómo personalizar el diseño de nuestra interfaz. Sin embargo, SdkTrays está diseñado para crear una interfaz **fácil** y **sencilla**, si queremos realizar una interfaz más compleja es mejor realizarlo con otra alternativa.

Figura A.4: SdkTrays facilita la construcción de interfaces sencillas.

En este ejemplo, todos los recursos de la escena han sido cargados desde un XML con la herramienta *DotSceneLoader*. Este fichero contiene las propiedades de cada nodo y se carga de manera dinámica en nuestra escena.

A.3.1. Sistema de Estados

Cada estado creado en nuestra aplicación incluye las siguientes acciones: entrar, pausar, resumir y salir. Para que todos los estados mantengan esta condición se utiliza **herencia** de la clase AppState. La clase que gestiona todos los estados es *AppStateManager* cuya tarea es manejarlos y **conectarlos**. Para hacer esto, el manager

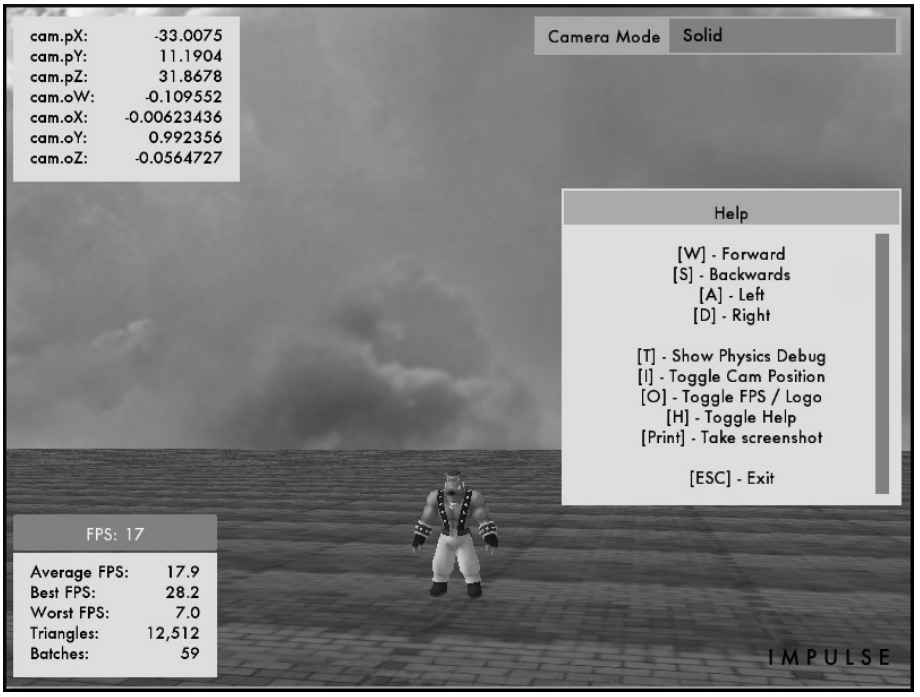


Figura A.3: Algunos de los widgets soportados en el menú del ejemplo.

tiene una pila de estados activos y siempre ejecuta el estado de la cima de la pila. En cualquier momento, se puede conectar otro estado de la pila, que será usado hasta que sea eliminado de la misma. En este caso, el manager resume el estado que dejó en pausa al introducir el anterior estado eliminado. En el momento en que la aplicación no contiene ningún estado activo, el manager cierra la aplicación.

A.3.2. Arquitectura

La arquitectura de *Advanced OgreFramework* parece compleja pero, realmente, es bastante fácil de entender. Esta herramienta gestiona la aplicación de Ogre como un conjunto de estados, que se van introduciendo en una pila. Esta gestión de estados es la que comúnmente hemos utilizado para los proyectos de Ogre.

OgreFramework

En la clase OgreFramework podemos observar que contiene todos los elementos básicos para iniciar una estancia de Ogre. OgreFramework contiene los siguientes gestores:

> Root, RenderWindow, ViewPort, Timer, Log, InputManager, SdkTraysManager.

Si queremos añadir más gestores como el gestor de sonido (*SDL, OpenAL, etc...*) debemos iniciarlos en esta clase.

El sistema se compone de una única instancia de Ogre y es accesible desde los estados de la aplicación. Para iniciar Ogre utilizaremos la función **initOgre()**, que crea los gestores necesarios para su funcionamiento como:

- Un gestor de registros LogManager.
- Crear el Root.
- Crea el RenderWindow y Viewport.
- Inicia los gestores de eventos de entrada mediante OIS.
- Carga los recursos e inicia el Timer.
- Establece el SDKTrayManager.

Para actualizar cada frame del motor Ogre, la clase *AppStateManager* llamará la función **updateOgre()** para actualizar directamente Ogre. Este método estará vacío para que cada estado se encargue de actualizar el motor de Ogre, pero es necesario ubicarlo aquí como tarea central del motor.

DotSceneLoader

Esta herramienta se usa para cargar las escenas en Ogre a partir de un fichero XML. Estos ficheros XML son procesados mediante *RapidXML*; *DotSceneLoader* crea las estructuras necesarias para ser cargadas en el *SceneManager* de Ogre. Si deseamos añadir físicas a un nodo, tendremos que añadir aquí su implementación.

Listado A.2: Carga de escena DotScene.

```
1 DotSceneLoader* pDotSceneLoader = new DotSceneLoader();
2 pDotSceneLoader->parseDotScene("CubeScene.xml", "General",
3         m_pSceneMgr, m_pSceneMgr->getRootSceneNode());
4 Ogre::Entity* cube = m_pSceneMgr->getEntity("Cube");
```

Podemos ver que para crear una escena es necesario introducir 4 argumentos: el fichero, el nombre del nodo, el *SceneManager* que se está gestionando en este estado y el nodo donde queremos introducirlo. Es muy interesante poder cargar las escenas desde un fichero XML por que podremos crearlas y destruirlas de manera dinámica, permitiendo una mejor gestión de la memoria y del nivel del juego.

Normalmente, se pueden agrupar las escenas de un nivel por bloques e ir cargando cada uno cuando sea necesario. Si, por ejemplo, nos encontramos con un juego tipo *endless-running*, es interesante ir destruyendo las escenas que quedan detrás del personaje ya que no vam a ser accesibles; también es interesante para ir auto-generando la escena de forma aleatoria hacia adelante.

Cada vez que carguemos una escena, podremos recuperar una entidad como se describe en el ejemplo; por un nombre bien conocido. Este nombre estará asignado en el fichero XML y podremos recuperar el nodo desde el propio juego. De esta manera podremos cambiar las propiedades o materiales de dicha entidad al vuelo como, por ejemplo, añadir un cuerpo rígido al motor de físicas y conectarlo a la entidad.

AppState

En este archivo se definen dos clases. La primera clase será la que herede el *AppStateManager* que gestiona los estados, pero está definido en esta clase por cuestiones de diseño. Esta contiene los métodos necesarios para la gestión de los estados, estos métodos son abstractos y están definidos en la clase *AppStateManager*.

La segunda clase será *AppState* la cuál será la clase de la que hereden todos los estados que se implementen en el juego. Esta clase contiene una serie de métodos para mantener la **consistencia entre estados** y poder tratar cada uno como estado independiente. En él se incluyen los métodos: enter, exit, pause, resume y update.

Estos métodos nos permitirán sobre todo mantener la interfaz organizada en cada estado como, por ejemplo, el track de música que se esté escuchando en cada estado. Cuando creamos un estado nuevo y lo agregamos a la pila de estados.

AppStateManager

Esta clase será la encargada de gestionar todos los estados. Tendrá los mecanismos para pausar, recuperar y cambiar un estado en cualquier instante del juego. Hereda directamente del *AppStateListener* e implementa sus métodos abstractos. Contiene dos vectores:

- Un vector para todos los estados existentes.
- Un vector para los estados activos.

Para cada estado existirán una serie de valores como su nombre, información y su estado. Cada vez que se cree un estado, se llamará al método **manageAppState()** para que introduzca dicha información al estado y lo inserte en la pila de estados existentes.

En esta clase se encuentra el **bucle principal del juego**, dentro del método **start()**. Dentro de este método podemos destacar el **cálculo del tiempo de un frame a otro**. Aquí podremos realizar las modificaciones necesarias si necesitamos coordinar el motor de renderizado con el motor de físicas. Para poder arreglar el problema en el desfase de tiempo entre ambos motores se ajusta un valor *delta* que discretiza el tiempo entre un frame y otro. Esta técnica nos ayudará a manejar correctamente el tiempo en ambos motores para que se comporte de manera natural.

Listado A.3: Bucle principal del juego.

```
double t = 0.0;
const double dt = 0.01;
double currentTime = tiempo_actual_en_segundos();
double accumulator = 0.0;
while ( !quit ) {
  double newTime = time(); // Tiempo actual en segundos
  double frameTime = newTime - currentTime; // Tiempo de un frame
  if ( frameTime > 0.25 )
    frameTime = 0.25;     // Tiempo maximo de espera entre frames
  currentTime = newTime;
  accumulator += frameTime;

  while ( accumulator >= dt ) {
    updateOgre(dt);
    accumulator -= dt;
  }

  const double alpha = accumulator / dt;
  renderOneFrame();
}
```

MenuState

La clase MenuState será una de las implementaciones de la clase *AppState*.Por herencia se asegura que todos los estados tienen los mismos métodos comunes y funciones como enter(), exit(), pause(), resume() y update(). Estos métodos se **ejecutarán automáticamente** por el *AppStateManager*. Esto es, si el estado actual fue pausado, ejecutará el método resume y seguirá la ejecución del bucle principal. Igual para el resto de estados. En todos los estados se escribirá la macro DECLARE-_APPSTATE_CLASS(MenuState) que insertará el nombre del estado para poder ser recuperado posteriormente.

Dentro de la función enter() de MenuState se crea un *SceneManager*, se crea la cámara, se construye la interfaz gráfica de usuario y se llama al creador de escena. En este caso el creador de escena no tendrá nada puesto que será un estado en el que se base en la interfaz gráfica creada por *SdkTrays* y no en Ogre.

En el método exit() estarán todos los destructores de escena, interfaz y cámara. De esta forma aseguramos la integridad entre estados. Los manejadores de entrada (teclado/ratón) también estarán presentes en todos los estados y servirán para capturar los eventos que se realicen en la interfaz.

La función update() actualiza los elementos de la interfaz como, por ejemplo, si el ratón está encima de un botón cambia la imagen; si, por otro lado, se pulsa el botón, la imagen cambia de nuevo. Además este método capturará si se ha enviado la orden de salir del sistema para finalizar la aplicación.

Y por último, el método buttonHit() es un *callback* que se accionará cuando se pulse un botón. Dentro de este método podremos identificar que botón se ha accionado y realizar una acción consecuente con ello.

GameState

Toda la lógica del juego está dentro del estado *GameState*. Aquí es donde **enlazaremos todos nuestros elementos de nuestro juego**, como por ejemplo: el motor de físicas, el controlador de nuestro personaje principal, módulos de inteligencia artificial, etc. En el caso de ejemplo de OgreFramework, incluye una escena mínima que carga una escena con *DotSceneLoader* y rellena de contenido la pantalla.

Para crear los elementos necesarios de la interfaz se invoca a builGUI() para insertar todos los elementos necesarios del estado mediante el gestor *SdkTraysManager* que explicaremos en la siguiente sección.

Para capturar los eventos de entrada por teclado, se utilizarán los métodos keyPressed() y keyReleased() para realizar los eventos correspondientes a si se pulsa una tecla o se ha dejado de pulsar respectivamente. En nuestro caso, tendremos una serie de condiciones para cada tecla que se pulse, las teclas son capturadas mediante las macros definidas en la librería OIS.

A.4. SdkTrays

Tanto los ejemplos que vienen con el SDK de Ogre (*SampleBrowser*), como OgreFramework utilizan SdkTrays. Este gestor de la interfaz se construyó con el propósito de ser **sencillo**, crear interfaces de una manera rápida y sin tanta complejidad como CEGUI. El sistema de SdkTrays está basado en Ogre::Overlay por lo que nos garantiza su **portabilidad a todas las plataformas**.

A.4.1. Introducción

El sistema de interfaz de SdkTrays se organiza en una serie de paneles (*trays*), que se localizan en nueve de posiciones de nuestra pantalla. Cada introducimos un nuevo *Ogre::Overlay* en nuestra escena, debemos calcular las coordenadas (*x,y*) del frame actual, añadiendo complejidad a la hora de diseñar una interfaz. Cuando creamos un *widget* nuevo con SdkTrays, debemos pasarle una posición de las nueve posibles y el tamaño. Si ya existe una un widget en esa región, el nuevo widget se colocará justo debajo, manteniendo las proporciones de la pantalla y redimiensionando el panel actual donde se encuentra dicho widget.

A.4.2. Requisitos

SdkTrays necesita la biblioteca OIS para la captura de datos de entrada mediante ratón o teclado. Una vez que se han incluido estas dependencias al proyecto simplemente es necesario incluir 'SdkTrays.h' en cada clase. Y también es necesario incluir el fichero 'SdkTrays.zip' a nuestro directorio de recursos, normalmente en /media/packs.

A.4.3. SdkTrayManager

Para utilizar SdkTrays, es necesario crear un *SdkTrayManager*. Esta es la clase a través de la cual se van a crear y administrar todos los *widgets*, manipular el cursor, cambiar la imagen de fondo, ajustar las propiedades de los paneles, mostrar diálogos, mostrar / ocultar la barra de carga, etc. El SdkTrayManager requiere "SdkTrays.zip", por lo que sólo se puede crear después de cargar ese recurso. Es recomendable asegurarse de que está utilizando el espacio de nombres OgreBites para poder incluir las macros de localización. Estas macros permiten crear un widget en las esquinas, en la parte central de la pantalla y en la parte central de los bordes.

A.4.4. Widgets

Existen 10 tipos de widgets básicos. Cada widget es **sólo un ejemplo de una plantilla** de OverlayElement. Cada vez que se crea un widget es necesario introducir las medidas en pixeles. **Cada widget está identificado con un nombre** y se gestionará su creación y destrucción mediante *SdkTrayManager*. Algunos de los widgets predefinidos que podemos crear son los siguientes:

> Button, TextBox, SelectMenu, Label, Separator, Slider, ParamsPanel, CheckBox, DecorWidget, ProgressBar, FrameStats, Loading Bar, Question Dialog.

A.4.5. SdkTrayListener

Esta clase contiene los controladores para todos los diferentes eventos que pueden disparar los widgets. La clase *SdkTrayManager* es, en sí misma, un *SdkTrayListener* porque responde a los eventos de los widgets. Algunos widgets dan la opción de no disparar un evento cuando cambia su estado. Esto es útil para inicializar o restablecer un widget, en cuyo caso no debe haber una respuesta de cualquier tipo.

A.4.6. Skins

SdkTrays no está orientado a una personalización profunda de los widgets como en CEGUI. Pero eso no significa que no podamos cambiar su apariencia del modelo estándar. Los recursos de las imágenes que utilizan los widgets se encuentran en el fichero *SdkTrays.zip*.

Si descomprimimos los recursos y modificamos el contenido, podemos personalizar la apariencia de los widgets. Las imágenes que se muestran tienden a ser muy pequeñas con formas muy concretas. Este tipo de imágenes se utiliza para reducir el espacio necesario para cada widget y que sea reutilizable por recursos. Si queremos modificar la apariencia de un widget es recomendable seguir las mismas formas y cambiar sólo los colores o también podemos optar por cambiar la apariencia a colores sólidos/planos. Las imágenes del paquete 'SdkTrays.zip' son PNG, por lo que podemos crear transparencias si lo deseamos o sombreados.

La personalización es limitada, pero con diseño y creatividad se pueden conseguir apariencias bastante diferentes de la original de OgreFramework.

A.4.7. OgreBites

Podemos observar como se distribuyen los widgets por regiones en la pantalla. Estas posiciones vienen dadas por el espacio de nombres OgreBites, la cual incluye una serie de macros que instancia los widgets en la región deseada. Observamos además que si por ejemplo agregamos seis widgets en la parte inferior derecha, los widgets se agrupan en un columna tipo pila. Este orden es el mismo que nosotros agregaremos en las lineas de código y las macros para instanciar los widgets son las siguientes:

TL_TOPLEFT, TL_TOPCENTER, TL_TOPRIGHT, TL_LEFT, TL_CENTER, TL_RIGHT, TL_BOTTOMLEFT, TL_BOTTOMCENTER, TL_BOTTOMRIGHT.

A.5. btOgre

btOgre es un **conector ligero** entre Bullet–Ogre. Este conector **no** proporciona un RigidBody ni CollisionShape directamente. En lugar de eso, se accede directamente al motor correspondiente. La única acción que realiza btOgre es la de **conectar** los *btRigidBodies* con *SceneNodes*. También puede convertir las mallas (*mesh*) de Ogre a figuras convexas en Bullet, proporcionando un mecanismo de *debug* de Bullet como observamos en la imagen y herramientas de conversión de un vector de Bullet a un quaternio de Ogre (ver Figura A.5).

En este ejemplo (listado A.4)comprobamos como btOgre conecta el Nodo de Ogre con la forma convexa de Bullet. Cada vez que se aplique un cambio en el mundo físico de Bullet **se verán sus consecuencias** en el Nodo de Ogre. Esto permite la integración de los dos motores y el acceso independiente a las propiedades del objeto en ambos mundos.

Cada vez que actualicemos el bucle principal del juego será necesario hacer una llamada a stepSimulation con el tiempo calculado entre frame y frame.

Listado A.5: Configuración de la escena.

```
1 // Creamos el objeto debug para Bullet
```

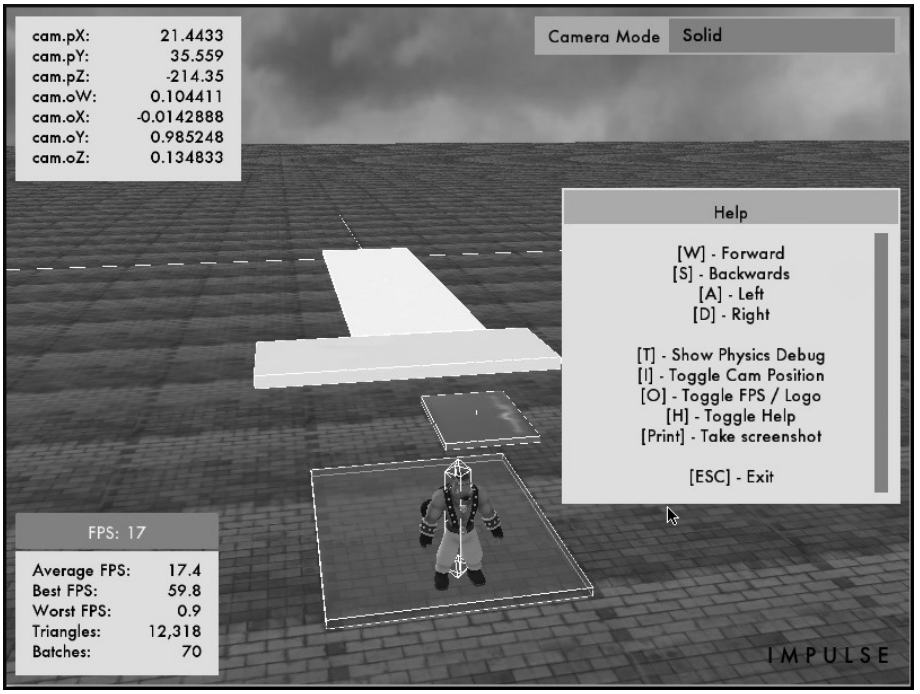


Figura A.5: Modo Debug con simulación física en btOgre.

```
BtOgre::DebugDrawer m_pDebugDrawer =
  new BtOgre::DebugDrawer(OgreFramework::getSingletonPtr()->
      m_pSceneMgr->getRootSceneNode(), m_pDynamicsWorld);
m_pDynamicsWorld->setDebugDrawer(m_pDebugDrawer);

// Actualizamos el motor Bullet
m_pDynamicsWorld->stepSimulation(timeSinceLastFrame);
m_pDebugDrawer->step();
```

Listado A.4: Configuración de Bullet.

```
1  //Creamos una Entidad y un Nodo en Ogre.
2  m_pSphereEntity = OgreFramework::getSingletonPtr()->
3    m_pSceneMgr->createEntity("SphereEntity", "Sphere.mesh");
4  m_pSphereNode = OgreFramework::getSingletonPtr()->
5    m_pSceneMgr->getRootSceneNode()->createChildSceneNode("SphereNode
         ");
6  m_pSphereNode->attachObject(m_pSphereEntity);
7
8  //Creamos la forma de la malla en Bullet.
9  BtOgre::StaticMeshToShapeConverter converter(m_pSphereEntity);
10 m_pSphereShape = converter.createSphere();
11
12 // Insertamos la masa al cuerpo rigido.
13 btScalar mass = 5;
14 btVector3 inertia;
15 m_pSphereShape->calculateLocalInertia(mass, inertia);
16
17 // Creamos un estado de BtOgre (conectamos Ogre y Bullet).
18 BtOgre::RigidBodyState *state = new BtOgre::RigidBodyState(
       m_pSphereNode);
19
20 // Creamos el Body.
21 BtOgre::btRigidBody m_pRigidBody =
22   new btRigidBody(mass, state, m_pSphereShape, inertia);
23 // Mundo fisico de Bullet (btDynamicsWorld).
24 m_pDynamicsWorld->addRigidBody(m_pRigidBody);
```

A.6. Referencias

- http://www.ogre3d.org/tikiwiki/Basic+Ogre+Framework
- http://www.ogre3d.org/tikiwiki/Advanced+Ogre+Framework
- http://bitbucket.org/spacegaier/basicogreframework
- http://bitbucket.org/spacegaier/advancedogreframework
- http://gafferongames.com/game-physics/fix-your-timestep/
- http://ogrees.wikispaces.com/Tutorial+Framework+Avanzado+de+Ogre
- http://www.ogre3d.org/tikiwiki/SdkTrays
- http://www.ogre3d.org/tikiwiki/OgreBites
- http://github.com/nikki93/btogre

Anexo B
Vault Defense al detalle

David Frutos Talavera

En este anexo se explicarán ciertos detalles del juego Vault Defense. Detalles de su GUI (Graphical User Interface), interfaz que usa CEGUI y con el cual, se han creado unos botones propios, una pantalla de carga y un efecto de cámara manchada de sangre, también se explicará al detalle el efecto de parpadeo de las luces, usado para darles un aspecto de luz de antorcha, y por último los enemigos, unos agentes de estado que implementan ciertos comportamientos.

Pero antes de explicar dichos detalles del juego, vamos a explicar cómo configurar Visual C++ 2010 Express [1] para poder compilar nuestros juegos con OGRE + CEGUI para la plataforma Windows.

B.1. Configuración en Visual 2010 Express

Este apartado es un "tutorial"paso a paso de cómo configurar bien un proyecto de Visual C++ para poder compilar nuestros juegos. Hay cosas que se dan por entendidas y otras que, aunque puedan parecer muy sencillas, se explicarán para no dejar ningún cabo suelto. Antes de nada lo primero es bajarse la versión correcta de CEGUI y OGRE, en el momento de escribir este apartado la versión de OGRE era la **1.8.1** y la **0.7.9** de CEGUI.

[1] 'http://www.microsoft.com/visualstudio/eng/downloads"

En la sección de “Downloads” de la Web de CEGUI [2] encontraremos la versión 0.7.9, de dicho apartado necesitamos descargar el código para Windows y las dependencias para MSVC++ 2010. Los descomprimimos donde queramos, eso sí, las **dependencias** para MSVC++ se tienen que colocar dentro de la carpeta de CEGUI-0.7.9.

Para OGRE solo necesitaremos descargar el SDK de OGRE[3]. En concreto, esta versión .ºGRE 1.8.1 SDK for Visual C++ .Net 2010 (32-bit)". Como ya ocurría con CEGUI este archivo se puede descomprimir donde queramos.

Después de esto se pueden configurar ciertos path en Windows, incluso buscar algún template para Visual C++, pero es mejor aprender cómo se configura bien un proyecto, y qué mejor manera de aprender que compilar las bibliotecas de CEGUI.

B.1.1. Compilando CEGUI en Windows

Para este paso iremos a la carpeta CEGUI-0.7.9/projects/premake. En dicha carpeta nos toca preparar el archivo .bat que vamos a ejecutar. Antes de nada, abriremos el archivo config.lua y buscaremos por la linea 119 esta sección:

```
-----------

-- Renderers

-- this controls which renderer modules are built
```

En este apartado del archivo podemos configurar qué proyectos queremos crear para Visual C++. Para ello, pondremos a **true** la variable **OGRE_RENDERER** . Con esto, el archivo estará bien configurado para nuestras necesidades, y aunque se puede configurar mejor para que ponga las rutas bien desde el principio es mejor aprender dónde se configura el proyecto en el programa. Ahora ejecutaremos el archivo *build_vs2008.bat* , aunque es un proyecto de 2008 se podrá cargar con Visual C++ 2010, se generará un archivo *CEGUI.sln*, dicho archivo será abierto con el Visual C++. cuando lo abramos, aceptaremos la conversión del proyecto sin tocar las opciones y ya podremos pasar a configurar las rutas.

Con el proyecto ya cargado y con la conversión ya realizada podemos pasar a configurar las rutas. Lo primero es saber que Visual C++ mantiene 2 tipos de configuraciones: una para modo debug y otra para modo Release, se pueden configurar muchas más, pero no será necesario. Comprobamos si están bien las rutas, que tienen que estarlo en el proyecto **CEGUIBase**, pulsaremos botón derecho/propiedades sobre el proyecto, en el explorador de soluciones y nos saldrá una ventana de configuración (ver figura B.1), o con el proyecto seleccionado Alt + Enter. Observamos que esta activo el modo Release, también hará falta comprobar las rutas para el modo Debug. Esto es lo que veremos:

1. Apartado C/C++ / general (Figura B.2). En dicho apartado se configuran los **“includes”** del proyecto, podemos observar que en CEGUIBase se añaden `../../../cegui/include`; y, por otra parte, `../../../dependencies/incluye`; por eso es necesario colocar la carpeta “dependencias” en la carpeta de CEGUI.

2. Vinculador / General (Figura B.3). Aquí es donde diremos dónde están nuestras **bibliotecas** (las carpetas Lib).

[2]’http://www.cegui.org.uk/wiki/index.php/Downloads

[3]’http://www.ogre3d.org/download/sdk

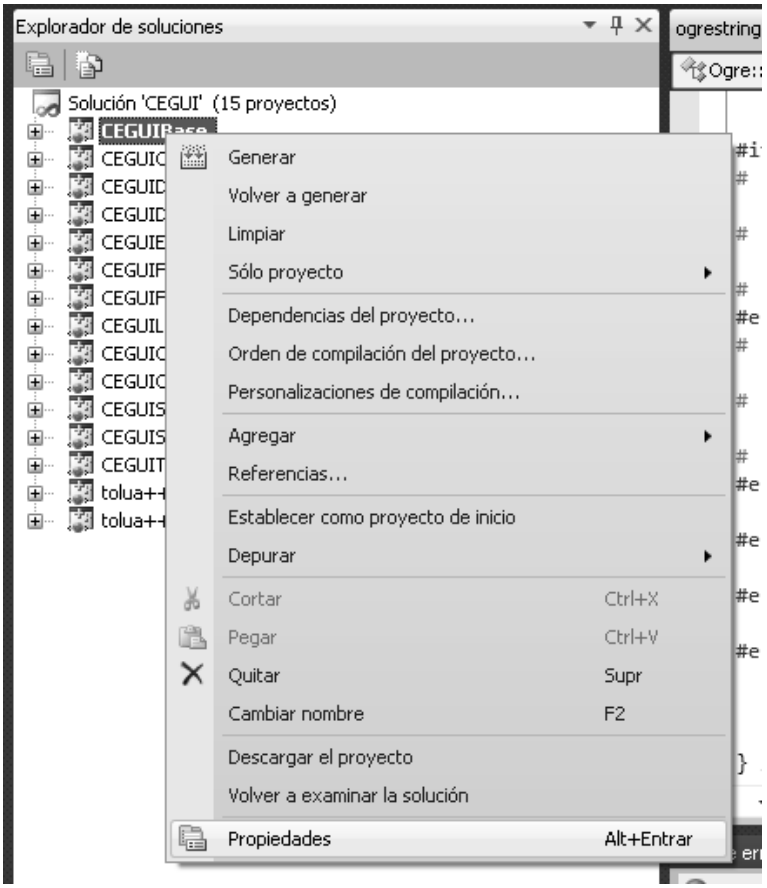


Figura B.1: Menú del proyecto.

3. Vinculador / Entrada (Figura B.4). En este apartado se colocan los nombre de las bibliotecas adicionales, es decir, las bibliotecas que estén en las rutas añadidas en general y que vayamos a usar.

Ahora lo siguiente es comprobar que **CEGUIOgreRenderer** está bien configurado repasando dichos apartados. Por defecto el config.lua tiene la ruta de Ogre pero mal puesta, tendremos que configurar todos los apartados. Al final quedará algo así, suponiendo que lo tenéis instalado en la raíz.

Con las rutas bien configuradas pasamos a compilar toda la solución, esto creara una serie de archivos en la carpeta CEGUI-0.7.9/lib y CEGUI-0.7.9/bin, algunos de estos archivos seran necesarios para poder compilar nuestros juegos.

B.1.2. Configurando OGRE en Windows

Explicado lo anterior pasamos a configurar Ogre en Windows.

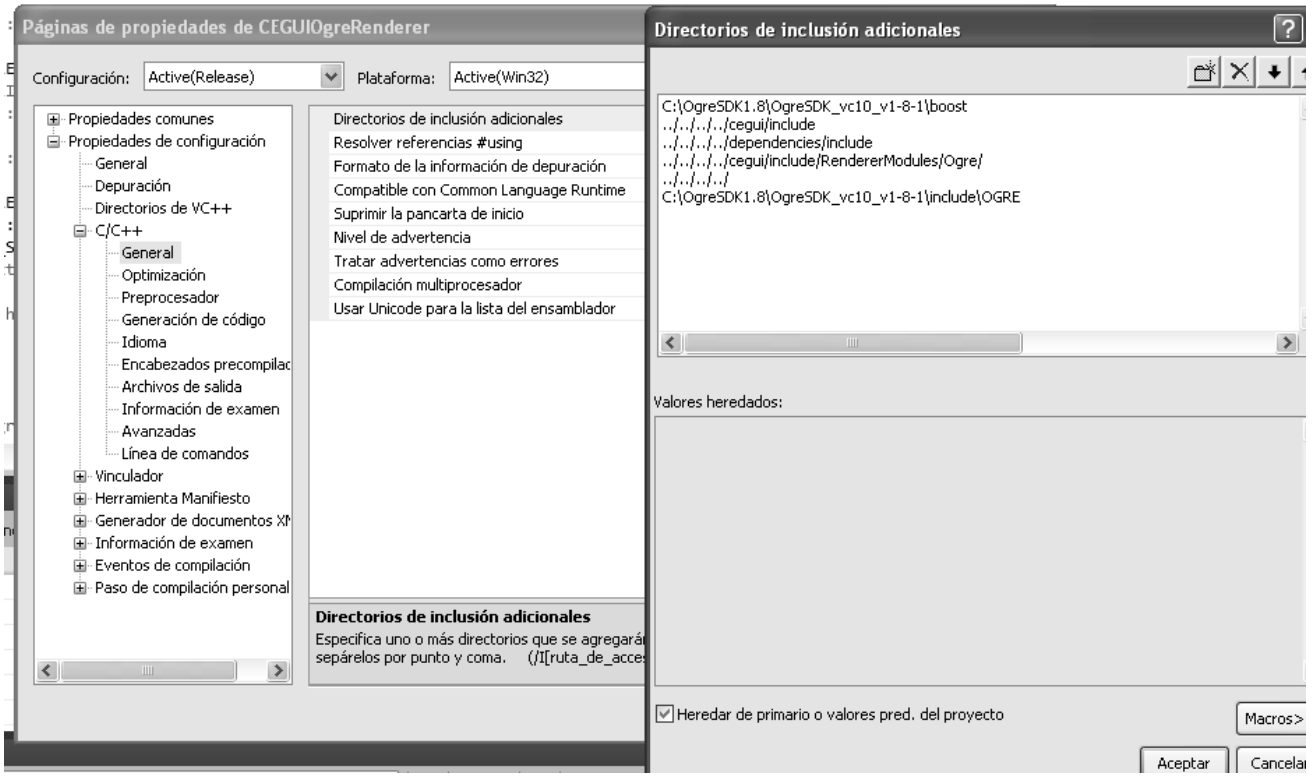


Figura B.2: Configuración de los directorios de inclusión. CEGUI.

Conociendo ya todos los apartados de la configuración del proyecto, solo será necesario enlazar bien todo, en includes añadiremos los includes de CEGUI y demás cosas que necesitemos (SDL, Xerces, etc...) y en el vinculador añadiremos las bibliotecas necesarias (ver figuras B.5, B.6 y B.7). Podemos ver la configuración necesaria para compilar el juego Vault Defense.

Una vez configurado el proyecto de OGRE pasamos a compilarlo. Recordad que aquí también tendremos una configuración release y una debug, es necesario saber cual estamos realizando para saber que .dlls necesitaremos para la ejecución del juego. También decir que necesitaremos compilar CEGUI para dicha versión.

Una vez compilado todo buscaremos la carpeta donde se ha generado el archivo ejecutable y añadiremos los archivos necesarios. Los principales son ogre.cfg, plugins.cfg y resources.cfg. El único que hace falta editar, si estamos usando uno de GNU/Linux, es el **plugins.cfg**. En la variable **PluginFolder** pondremos la ruta de la carpeta que contendra los .dlls de los plugins de OGRE, en nuestor caso la hemos llamado **plugins** que será la carpeta que tendremos que crear en el directorio del juego. Además comprobaremos que estamos cargando los plugins correctos; si hemos compilado todo en modo **debug** tenemos que cargar los plugins correctos añadiendo **_d** al nombre.

Un ejemplo de archivo para un juego compilado en modo debug sería éste.

```
# Directorio donde están los plugins
PluginFolder=plugins

Plugin=RenderSystem_GL_d
```

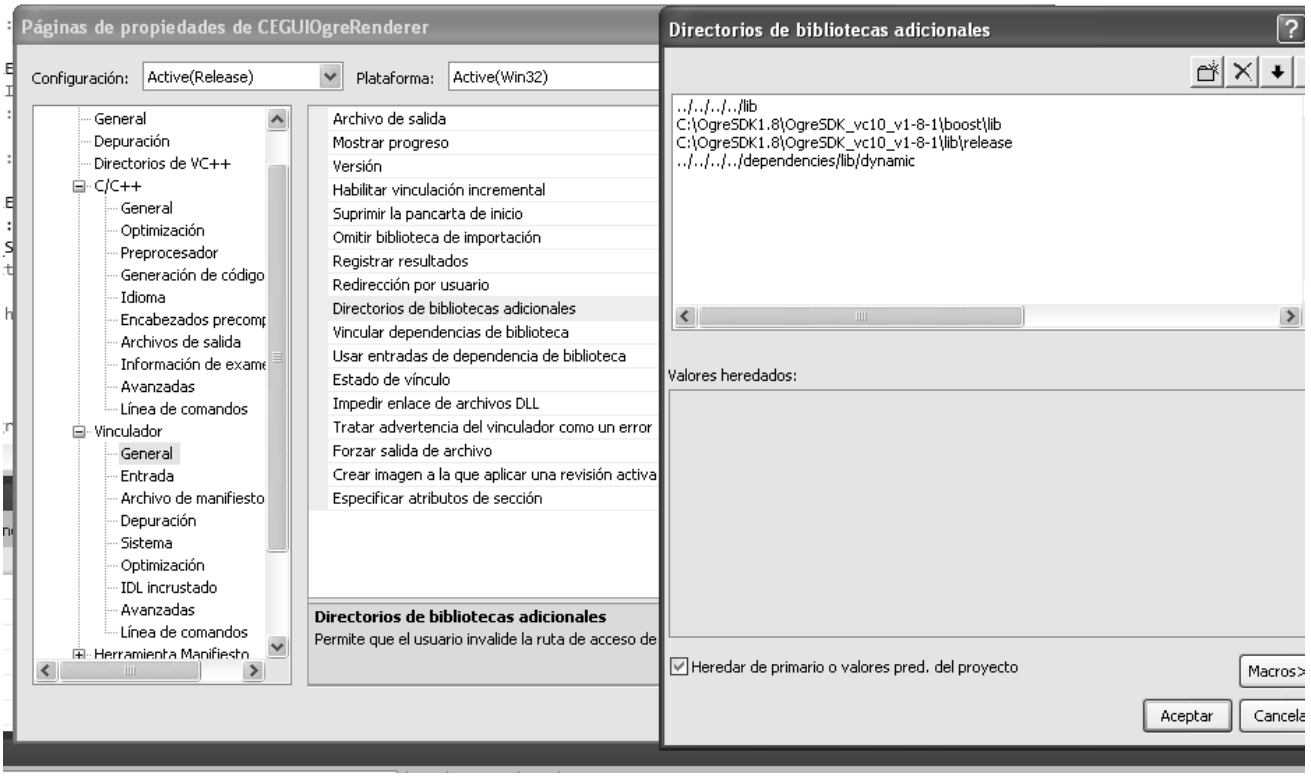


Figura B.3: Configuración de las bibliotecas adicionales. CEGUI.

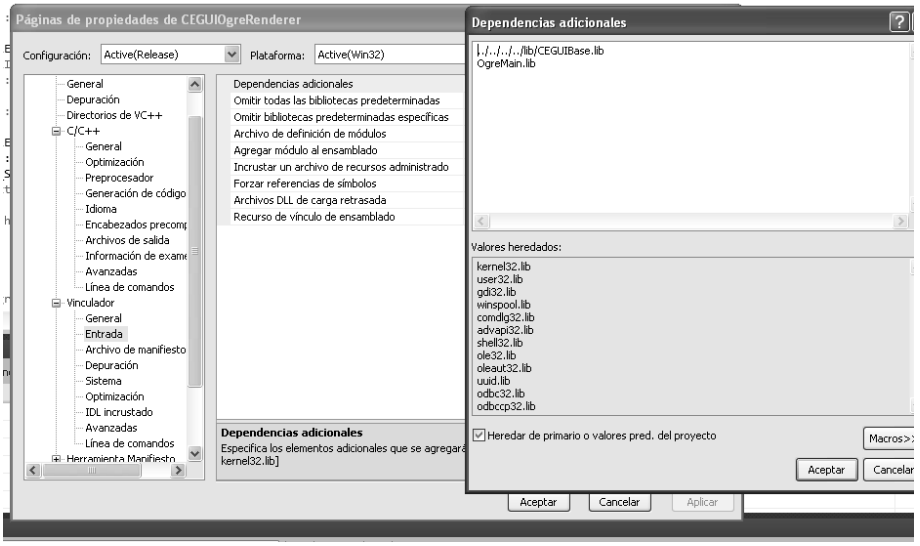


Figura B.4: Configuración de las dependencias adicionales. CEGUI.

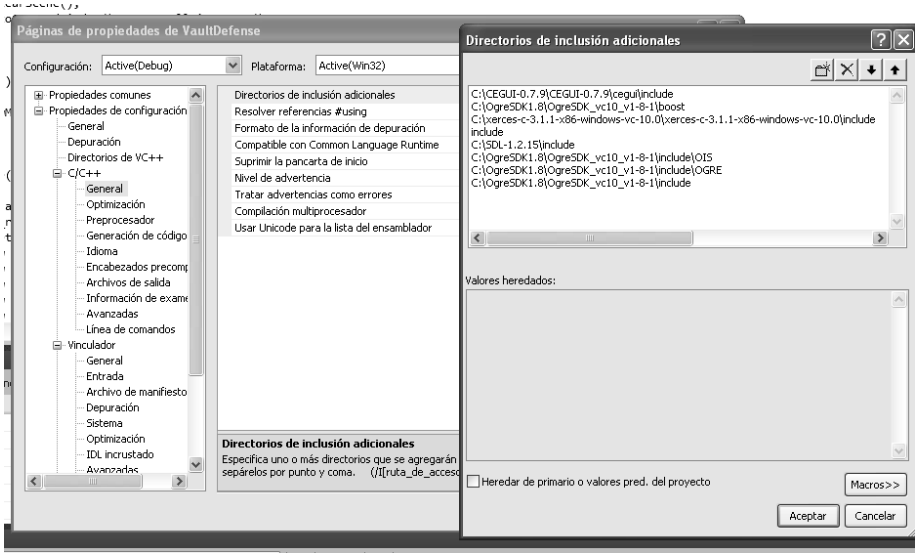


Figura B.5: Configuración de los directorios de inclusión. OGRE

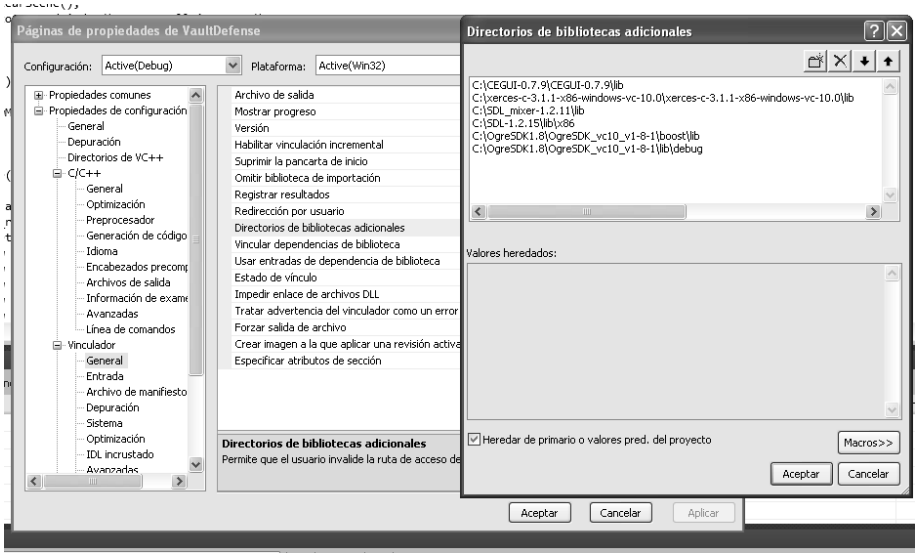


Figura B.6: Configuración de las bibliotecas adicionales. OGRE

```
Plugin=Plugin_ParticleFX_d
```

Ya con estos 3 archivos añadidos solo falta crear una carpeta plugins y añadir los .dlls correspondientes dentro de ella.

Con los plugins colocados pasamos a añadir los .dlls que faltan. Estos varían según las bibliotecas usadas pero por lo general hará falta **OgreMain** y **OIS**. Acordaros de copiar los dlls correspondientes según vuestra compilación, si estáis en modo **debug** usad los que terminan en **_d**.

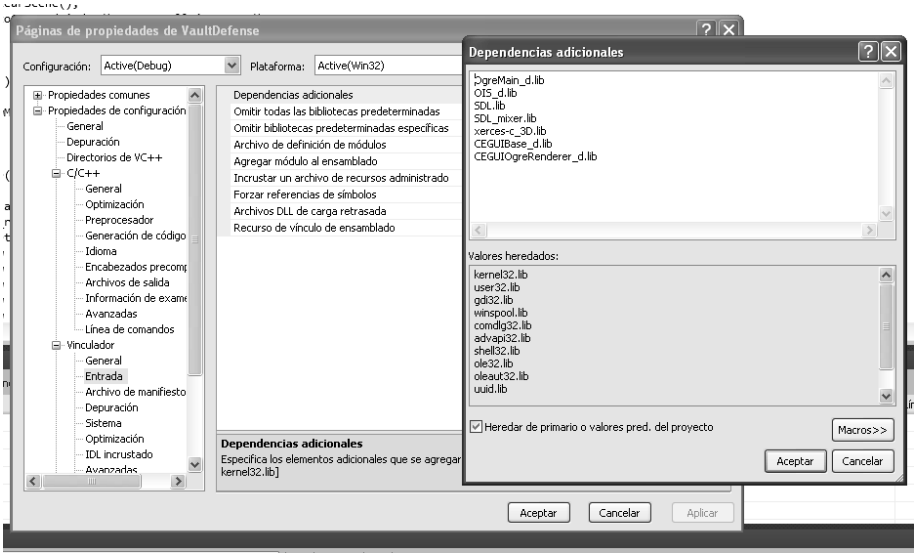


Figura B.7: Configuración de las dependencias adicionales. OGRE

Explicado esto ya podemos pasar a los detalles de Vault Defense.

B.2. GUI en Vault Defense

En Vault Defense se han creado unos botones propios, una pantalla de carga y un efecto de salpicadura en cámara, todo esto realizado con CEGUI.

B.2.1. Botones propios

Aunque CEGUI tiene un botón ya definido, no es un botón muy bonito, es el tipico botón que se puede usar para un formulario, pero no el botón que se usaria en un menu inicial de un juego, para estos casos lo mejor es crearse un botón propio. Para esta solucion es necesario crear archivos scheme, imageset y lookNFeel.

Crearemos un archivo .imageset propio, creando una imagen compuesta con todos los botones necesarios.

Ahora la parte complicada, el **LookNFeel**, para este fichero podemos adaptar un fichero ya existente, en los botones del Vault Defense se ha usado el archivo de TaharezLook.looknfeel.

Listado B.1: Fichero TaharezLook.looknfeel

```
<!-- ..................... -->
<WidgetLook name="VaultDefense/BotonSinglePlayer">
  <!-- ..................... -->
  <ImagerySection name="normal">
    <ImageryComponent>
      <Area>
    <Dim type="LeftEdge"><AbsoluteDim value="0" /></Dim>
    <Dim type="TopEdge"><AbsoluteDim value="0" /></Dim>
    <Dim type="Width"><UnifiedDim scale="1.0" type="Width" />
    </Dim>
    <Dim type="Height"><UnifiedDim scale="1.0" type="Height" />
    </Dim>
      </Area>
      <Image imageset="VaultDefense" image="BotonIzq" />
      <VertFormat type="Stretched" />
    </ImageryComponent>

    <ImageryComponent>
      <Area>
    <Dim type="LeftEdge"><AbsoluteDim value="0" /></Dim>
    <Dim type="TopEdge"><AbsoluteDim value="0" /></Dim>
    <Dim type="Width"><UnifiedDim scale="1" type="Width"/>
    </Dim>
    <Dim type="Height"><UnifiedDim scale="1" type="Height"/>
    </Dim>
      </Area>
      <Image imageset="VaultDefense" image="BotonDer" />
      <VertFormat type="Stretched" />
      <HorzFormat type="RightAligned" />
    </ImageryComponent>

    <ImageryComponent>
      <Area>
    <Dim type="LeftEdge">
      <ImageDim  imageset="VaultDefense"  image="BotonIzq"
            dimension="Width"/>
    </Dim>
    <Dim type="TopEdge"><AbsoluteDim value="0" /></Dim>
    <Dim type="RightEdge">
      <UnifiedDim scale="1" type="Width">
        <DimOperator op="Subtract">
          <ImageDim imageset="VaultDefense"
            image="BotonDer" dimension="Width"/>
```

```
44          </DimOperator>
45        </UnifiedDim>
46      </Dim>
47      <Dim type="Height"><UnifiedDim scale="1"
48      type="Height" /></Dim>
49        </Area>
50        <Image imageset="VaultDefense"
51           image="BotonSinglePlayerNormal" />
52        <VertFormat type="Stretched" />
53        <HorzFormat type="Stretched" />
54      </ImageryComponent>
55    </ImagerySection>
56
57    <ImagerySection name="hover">
58      <!--    ..................... -->
59    </ImagerySection>
60
61    <ImagerySection name="pushed">
62      <!--    ..................... -->
63    </ImagerySection>
64
65    <ImagerySection name="disabled">
66      <!--    ..................... -->
67    </ImagerySection>
68    <StateImagery name="Normal">
69      <Layer>
70        <Section section="normal" />
71      </Layer>
72    </StateImagery>
73    <StateImagery name="Hover">
74      <Layer>
75        <Section section="hover" />
76      </Layer>
77    </StateImagery>
78    <StateImagery name="Pushed">
79      <Layer>
80        <Section section="pushed" />
81      </Layer>
82    </StateImagery>
83    <StateImagery name="PushedOff">
84      <Layer>
85        <Section section="hover" />
86      </Layer>
87    </StateImagery>
88    <StateImagery name="Disabled">
89      <Layer>
90        <Section section="disabled">
91      <Colours topLeft="FF7F7F7F" topRight="FF7F7F7F"
92           bottomLeft="FF7F7F7F" bottomRight="FF7F7F7F" />
93        </Section>
94      </Layer>
95    </StateImagery>
96  </WidgetLook>
```

Lo importante de estos archivos es:

1. WidgetLook...: Se define el nombre del lookNFeel, este identificador se usará en el scheme.
2. ImagerySection... : Según el TargetType y el Renderer del archivo scheme estas secciones tienen un identificador en caso de los botones son:
   a) normal: Botón normal, sin nada que lo altere.
   b) hover: Botón con el ratón encima.
   c) pushed: Botón pulsado.
   d) disable: Botón “apagado”.

Dentro de ImagerySection podemos encontrar:

1. ImageryComponent...: Crea un componente a “dibujar”.
2. Area...: Define un área, definida por <Dim>
3. Image...: Define una imagen que irá dentro del ImageryComponent

Éstas son las marcas más usadas. Con estas marcas podemos crear unos botones propios que actúen como nosotros queremos. En el caso del juego Vault Defense los botones están creados por 3 zonas, una zona central donde se muestra la imagen del botón, y 2 zonas, una a cada lado, que se usara en caso de que el botón este en estado hover o pushed. La imagen BotonDer, BotonIzq es una imagen transparente que solo se usa para rellenar el hueco, en los estados hover y pushed se pone una imagen distinta dándole así un efecto curioso a los botones.

El codigo de hover, pushed y disable es similar al de normal, solo que cambia las imagenes que carga. En hover en los bloques laterales, los que en normal son BotonDer y BotonIzq cambian por otra imagen, en el caso del Vault Defense por BotonDerHover y BotonIzqHover, imagenes que muestran un pico y una ballesta, que apareceran a los lados del botón seleccionado.

Y lo último es el archivo scheme:

Listado B.2: Fichero TaharezLook.scheme

```
<GUIScheme Name="VaultDefense">
  <Imageset Filename="VaultDefense.imageset" />
  <LookNFeel Filename="VaultDefense.looknfeel" />
  <WindowRendererSet Filename="CEGUIFalagardWRBase" />
  <FalagardMapping WindowType="VaultDefense/BotonSinglePlayer"
            TargetType="CEGUI/PushButton" Renderer="Falagard/Button"
            LookNFeel="VaultDefense/BotonSinglePlayer" />
  <FalagardMapping WindowType="VaultDefense/BotonInstrucciones"
            TargetType="CEGUI/PushButton" Renderer="Falagard/Button"
            LookNFeel="VaultDefense/BotonInstrucciones" />
  <FalagardMapping WindowType="VaultDefense/BotonCreditos"
            TargetType="CEGUI/PushButton" Renderer="Falagard/Button"
            LookNFeel="VaultDefense/BotonCreditos" />
  <FalagardMapping WindowType="VaultDefense/BotonRecords"
            TargetType="CEGUI/PushButton" Renderer="Falagard/Button"
            LookNFeel="VaultDefense/BotonRecords" />
  <FalagardMapping WindowType="VaultDefense/BotonSalir"
            TargetType="CEGUI/PushButton" Renderer="Falagard/Button"
            LookNFeel="VaultDefense/BotonSalir" />
  <FalagardMapping WindowType="VaultDefense/StaticImage"
            TargetType="DefaultWindow" Renderer="Falagard/
                StaticImage"
            LookNFeel="VaultDefense/StaticImage" />
</GUIScheme>
```

Definimos que imageset vamos a usar, que looknfeel usamos, y luego pasamos a crear los botones con:

Listado B.3: Creación de botones.

```
<FalagardMapping WindowType="VaultDefense/BotonCreditos"
          TargetType="CEGUI/PushButton" Renderer="Falagard/Button"
          LookNFeel="VaultDefense/BotonCreditos"/>
```

En este código podemos ver la creación de unos botones muy simples, se ha cogido como base un archivo scheme ya creado y se ha modificado. En WindowType ponemos el identificador del window para poder usarlo en nuestros layouts y en LookNFeel ponemos nuestro lookNFeel modificado.

En el código es necesario cargar nuestros propios archivos:

Listado B.4: Carga de archivos.

```
1 CEGUI::SchemeManager::getSingleton().create
2         ("VaultDefense.scheme");
3
4 CEGUI::ImagesetManager::getSingleton().create
5         ("VaultDefense.imageset");
```

B.2.2. Pantalla de Carga

Para crear las pantallas de carga en Ogre es necesario conocer la función **renderOneFrame()**[4] de Ogre. Esta función obliga a Ogre a renderizar un solo frame; llamaremos a dicha función dentro de la función enter del estado donde se quiera añadir el cargador. Lo más normal es que sea al comienzo del juego, en la función enter del estado juego.

Listado B.5: Función enter() (clase PlayState).

```
1  void PlayState::enter () {
2
3    _root = Ogre::Root::getSingletonPtr();
4    _sceneMgr = _root->getSceneManager("SceneManager");
5
6    CEGUI::System::getSingleton().setDefaultMouseCursor
7      ("VaultDefense", "BotonDer");
8    CEGUI::Window *guiRoot =
9      CEGUI::WindowManager::getSingleton().loadWindowLayout
10     ("carga.layout");
11   CEGUI::System::getSingleton().setGUISheet(guiRoot);
12   CEGUI::DefaultWindow* staticText = static_cast<CEGUI::
13     DefaultWindow*>(CEGUI::WindowManager::getSingleton().
14               getWindow("carga/texto"));
15   staticText->setText("Generando mundo aleatorio 1-5");
16
17   CEGUI::ProgressBar* progressBar =
18     static_cast<CEGUI::ProgressBar*>
19     (CEGUI::WindowManager::getSingleton().
20      getWindow("carga/barra"));
21   _Renderizando=false;
22   _root->renderOneFrame();
23
24   /* ... */
25 }
```

La penúltima linea, el _renderizando=false, se usa para no ejecutar el código de la función **frameStarted(...)**, tened en cuenta que al ejecutar RenderOneFrame se estara llamando a frameStarted y es posible que algunas de las acciones que realizamos dentro de dicha función no se puedan realizar.

[4]'http://www.ogre3d.org/docs/api/html/classOgre_1_1Root.html

Este método de carga no calcula el tiempo real de lo que tardan las operaciones, es una carga por pasos. Se definen una serie de cargas, y segun se van cargando se va pasando de paso. En resumen, se dividen los calculos en grupos y se realiza un RenderOneFrame por cada calculo realizado, cuando se realiza el calculo se ejecuta este código.

Listado B.6: Renderizando frame a frame.

```
progressBar->step();
staticText->setText("Calculando caminos 2-5");
_root->renderOneFrame();
```

El step de la barra está definido en el layout y su valor es $1/n^{o}grupos$:

Listado B.7: GUI Layout.

```
<GUILayout >
  <Window Type="DefaultWindow" Name="carga" >
    <Property Name="UnifiedAreaRect"
          Value="{{0.0025,0},{0,0},{1.0025,0},{1,0}}" />
    <Window Type="VaultDefense/StaticImage"
            Name="carga/Fondo" >
      <Property Name="FrameEnabled" Value="False" />
      <Property Name="UnifiedAreaRect"
        Value="{{0,0},{0,0},{1,0},{1,0}}" />
      <Property Name="BackgroundEnabled" Value="False" />
      <Property Name="Image"
        Value="set:VaultDefenseFondo image:Fondo" />
      <Window Type="TaharezLook/ProgressBar"
          Name="carga/barra" >
    <Property Name="StepSize" Value="0.20" />
    <Property Name="CurrentProgress" Value="0.0" />
    <Property Name="UnifiedAreaRect"
          Value="{{0.0975004,0},{0.753333,0},
             {0.869998,0},{0.803333,0}}" />
   </Window>
   <Window Type="TaharezLook/StaticText"
           Name="carga/texto" >
     <Property Name="Font" Value="VaultDefense-15" />
     <Property Name="UnifiedAreaRect"
           Value="{{0.13875,0},{0.708333,0},
              {0.828753,0},{0.75,0}}" />
     <Property Name="BackgroundEnabled" Value="False" />
     <Property Name="FrameEnabled" Value="False" />
   </Window>
    </Window>
  </Window>
</GUILayout>
```

Esto se puede cambiar, se puede crear una barra propia y hacerle un escalado. También se puede poner una imagen en el centro e ir cambiándola. Existen muchas maneras, pero ésta es la más sencilla.

B.2.3. Manchas en la cámara

El método creado para crear manchas de sangre en la cámara es muy sencillo. Lo primero es añadir las imagenes al layout del juego, como una imagen normal,

Listado B.8: Imágenes en el Layout.

```
<Window Type="VaultDefense/StaticImage"
```

```
2          Name="UI/Fondo/Sangre10"><Property Name="Image"
3          Value="set:VaultDefenseSangre2 image:Sangre"/>
4  <Property Name="UnifiedAreaRect"
5        Value="{{0.0,0},{0.0,0},{1,0},{1,0}}"/>
6  <Property Name="BackgroundEnabled" Value="False"/>
7  <Property Name="Alpha" Value="0"/>
8  <Property Name="FrameEnabled" Value="False"/>
9  </Window>
```

Después se crea una estructura de datos para almacenar el nombre de la imagen, su variable alpha y si está activa o no.

Listado B.9: Estructura sangreCEGUI.

```
1 typedef struct {
2   String nombre;
3   double alpha;
4   bool activo;
5 } sangreCEGUI;
```

Cuando estemos creando la interfaz pasamos a poner el alpha de todas las imágenes a 0.

Listado B.10: Estableciendo el valor de alpha a 0

```
1 for (int i = 1; i < 11; i++) {
2   std::stringstream auxNombre;
3   auxNombre << "UI/Fondo/Sangre" << i;
4   sangreCEGUI aux;
5   aux.nombre=auxNombre.str();
6   aux.alpha=0.0;
7   aux.activo=false;
8   _sangrado.push_back(aux);
9 }
```

Luego en alguna parte del juego, en el caso de Vault Defense cuando el personaje recibe un impacto, activamos una de las sangres de manera aleatoria, mostrándola y poniendo su alpha a 1, durante el enterFrame() vamos decrementando el alpha.

Listado B.11: Activando la sangre.

```
1  for(int i = 0; i < _sangrado.size(); i++) {
2
3    if(_sangrado[i].activo) {
4      CEGUI::DefaultWindow* staticImage =
5        static_cast<CEGUI::DefaultWindow*>
6        (CEGUI::WindowManager::getSingleton().
7         getWindow(_sangrado[i].nombre));
8
9      _sangrado[i].alpha-=0.5*deltaT;
10
11     std::stringstream auxS;
12     auxS << _sangrado[i].alpha;
13     staticImage->setProperty("Alpha",auxS.str());
14
15     if(_sangrado[i].alpha<=0.0) {
16       _sangrado[i].activo=false;
17     }
18
19   }
20
21 }
```

Esto creara un efecto de salpicon de sangre, creando una mancha en la pantalla y haciendo que se desvanezca gradualmente.

B.3. Luces en Vault Defense: efecto fuego

El efecto de luz de fuego hace uso de la clase **Controller y ControllerValue** de Ogre, que nos facilita el poder variar los valores de color e intensidad de la luz, simulando la llama del fuego. Se ha añadido también una pequeña variación en el movimiento del punto de luz para simular el movimiento de las sombras producido por el fuego.

Para dicho efecto se uso el código que se muestra en el siguiente listado.

Listado B.12: Activando la sangre.

```
1  Ogre::Light* fuego = _sceneMgr->createLight("Fuego");
2  fuego->setType(Ogre::Light::LT_POINT);
3  fuego->setDiffuseColour(0.8,0.5,0.0);
4
5  Ogre::SceneNode* nodeFuego =
6    _sceneMgr->createSceneNode("nodeFuego");
7  nodeFuego->attachObject(fuego);
8  nodeFuego->setPosition(314,2,290);
9
10 ControllerValueRealPtr RedLightFlasher =
11   ControllerValueRealPtr(new LightBlinker
12               (nodeFuego,fuego, ColourValue(0.8,0.5,0.0),
13                ColourValue(0.6,0.4,0.0), 0.975));
14 ControllerFunctionRealPtr RedLightControllerFunc =
15   ControllerFunctionRealPtr(new LightFlasherControllerFunction
16                 (Ogre::WFT\_SINE, 3.50, 0.0));
17 _RedLightController = _ControllerManager->createController
18   (_ControllerManager->getFrameTimeSource(),
19    ControllerValueRealPtr(RedLightFlasher),
20    ControllerFunctionRealPtr(RedLightControllerFunc));
21
22 RedLightFlasher.freeMethod();
23 RedLightControllerFunc.freeMethod();
```

ControllerValueRealPtr es un controlador de variables a actualizar, basicamente lo que recibe es un objeto que hereda de ControllerValue<T>, dicho objeto tiene que tener un método setValue y un getValue.

En la instrucción siguiente tenemos la función que va a calcular la variable, LightFlasherControllerFuntion. Esto devuelve un valor entre 0 y 1 que se usará en el setValue de la línea anterior.

En la línea siguientes lo que se hace es configurar bien el controlador para que obtenga el tiempo entre frames y que sepa que controladores usar y que objetos modificar.

Es un poco complejo de explicar y menos en un par de líneas, pero para mas informacion se puede visitar la web oficial de Ogre[5].

[5] http://www.ogre3d.org/tikiwiki/tiki-index.php?page=LightsCameraAction

B.4. Enemigos en Vault Defense

Bueno la explicación de los enemigos en sí es muy extensa, todo esta entrelazado. Esta explicación irá detallando el cómo según se vaya necesitando.

Inicialmente los enemigos están creados como un agente de estados ya explicado en el curso. La transición entre dichos estados se realiza dependiendo de la proximidad al jugador y al tesoro.

Dichos estados se muestran visualmente gracias a un icono posicionado encima de los enemigos.

Listado B.13: Gestión de la IA.

```
1 _ia = sceneMgr->createBillboardSet(billAux.str(),1);
2 _ia->setBillboardType(Ogre::BBT_POINT);
3 _ia->setMaterialName("IATesoro");
4 _ia->setDefaultDimensions(1.,1);
5 _ia->createBillboard(Ogre::Vector3(0,0,0));
6 Ogre::SceneNode* iaNode =
7   _modelo->createChildSceneNode(billNodeAux.str());
8 iaNode->setPosition(0,6.0,0);
9 iaNode->attachObject(_ia);
```

Cuando el estado cambia se realiza además una invocación a la función *setMaterialName()*, que cambia la imagen a mostrar.

Lo más importante de los enemigos es su forma de guiarse por el escenario; esto se puede explicar respondiendo a una serie de preguntas.

¿Cómo se mueven los enemigos?

Bueno, para el movimiento de los enemigos se usó un pathfinder, un algoritmo A-star para ser más concreto. Dicho algoritmo devuelve un camino a los enemigos para que recorran los pasillos del juego sin problemas, dicho camino es devuelto en puntos y los enemigos van de punto a punto, encarando al punto y trasladándose en su dirección.

¿Cómo se evitan las paredes, trampas y columnas ?

En la creación del escenario se creo un array tridimensional para almacenar cierta información (trampas y bloques). Dicho array se relleno usando unos archivos xml generados mediante un script de Blender; en dicho array viene la posicion de los bloques que forman el escenario. Gracias a esto y cogiendo lo que viene a ser el suelo de array se puede rellenar un mapa del suelo que se le pasa al pathfinder. De este modo, las trampas se guardan en el mapa y se puede calcular nuevas rutas dependiendo de la inteligencia de los enemigos.

¿Como evitan chocarse los enemigos?

Los enemigos llevan a sus pies unos bloques pequeños ocultos. Dichos bloques se usan para calcular colisiones entre cajas **AABB.**]

Listado B.14: Evasión de colisiones.

```
1 Vector3 trans(0,0,0);
```

```
2  for (i =_ rMn; i < _ID; i++) {
3    std::stringstream saux, saux2;
4
5    if(_ID!=i) {
6      saux << "nodeAABBEnemigo" << i;
7      saux2 << "nodeAABBEnemigo" << _ID;
8      Ogre::SceneManager *sceneMgr =
9        Ogre::Root::getSingletonPtr()->
10       getSceneManager("SceneManager");
11     Ogre::SceneNode *nodeAux = sceneMgr->
12       getSceneNode(saux.str());
13     Ogre::SceneNode *nodeAct = sceneMgr->
14       getSceneNode(saux2.str());
15     AxisAlignedBox aab = nodeAct->_getWorldAABB().
16       intersection(nodeAux->_getWorldAABB());
17
18     if(!aab.isNull()) {
19       Vector3 diff = nodeAux->getPosition() -
20     _modelo->getPosition();
21       Vector3 dir = diff.normalisedCopy();
22       Vector3 pen = aab.getMaximum() - aab.getMinimum();
23       trans = dir *
24     Math::Abs(pen.normalisedCopy().dotProduct(dir)) *
25     pen.length();
26     }
27
28   }
29
30 }
31
32 _modelo->translate(-trans);
```

La parte importante son las líneas 15-16. Dicha función comprueba la intersección de 2 nodos y devuelve una AxixAlignedBox, si es distinto de Null la caja invisible esta chocando con otra caja entonces ambas cajas se repelen.

Bibliografía

[1] www.boost.org.

[2] www.cegui.org.uk.

[3] www.swig.org.

[4] ISO/IEC 9241. *ISO/IEC 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on usability*. ISO, 1998.

[5] Advanced Micro Devices, Inc., Disponible en línea en `http://support.amd.com/us/Processor_TechDocs/31116.pdf`. *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors*, Apr. 2010.

[6] T. Akenine-Moller, E. Haines, and N. Hoffman. *Real-Time Rendering (3rd Edition)*. AK Peters, 2008.

[7] E. Akenine-Möller, T. Haines and N. Hoffman. *Real-Time Rendering*. AK Peters, Ltd., 2008.

[8] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering*. AK Peters, 3^{rd} edition, 2008.

[9] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 2001.

[10] Grenville Armitage, Mark Claypool, and Philip Branch. *Networking and Online Games*. Wiley, 2006.

[11] K. Beck. *Extreme Programming Explained: Embrace Change. Addison- Wesley Professional*. Addison-Wesley Professional, 1999.

[12] E Bethke. *Game Development and Production*. Wordware Publishing, 2003.

[13] Carlos Ble. *Diseño ágil con TDD*. 2009.

[14] D.M. Bourg and G. Seemann. *AI for game developers*. O'Reilly Media, 2004.

[15] Gary Bradski and Adrian Kaehler. *Learning OpenCV*. O'Reilly Media Inc., 2008.

[16] M. Buckland. *Programming Game AI by Example*. Wordware Game Developer's Library, 2004.

[17] D. Bulka and D. Mayhew. *Efficient C++, Performance Programming Techniques*. Addison-Wesley, 1999.

[18] Linux Kernel Community. *Git Tutorial*. 2013.

[19] James O. Coplien. Curiously recurring template patterns. *C++ Report*, February 1995.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, third edition edition, 2009.

[21] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. 2009.

[22] Valve Corporation. http://source.valvesoftware.com/.

[23] Chris Dallaire. Binary triangle trees for terrain tile index buffer generation. 2006.

[24] D.S.C. Dalmau. *Core techniques and algorithms in game programming*. New Riders Pub, 2004.

[25] W.H. De Boer. Fast terrain rendering using geometrical mipmapping. *Unpublished paper, available at http://www. flipcode. com/articles/article geomipmaps. pdf*, 2000.

[26] X. Décoret, F. Durand, F.X. Sillion, and J. Dorsey. Billboard clouds for extreme model simplification. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 689–696. ACM, 2003.

[27] M.J. Dickheiser. *C++ for Game Programmers (2nd Edition)*. Charles River Media, 2007.

[28] A. Dix. *Human computer interaction*. Prentice Hall, 1993.

[29] M. Duchaineau, M. Wolinsky, D.E. Sigeti, M.C. Miller, C. Aldrich, and M.B. Mineev-Weinstein. Roaming terrain: real-time optimally adapting meshes. In *Visualization'97., Proceedings*, pages 81–88. IEEE, 1997.

[30] D.H. Eberly. *3D Game Engine Architecture*. Morgan Kauffmann, 2005.

[31] C. Ericson. *Real-time collision detection*, volume 1. Morgan Kaufmann, 2005.

[32] G. Farin and D. Hansford. *Practical Linear Algebra: A Geometry Toolbox*. CRC Press, 2013.

[33] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[34] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[35] K. Flood. Game unified process (gup).

[36] S. Franklin and Graesser A. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*, pages 21–35, London, UK, 1997. Springer-Verlag.

[37] E. Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[38] E. Gamma, R. Helm, R. Johnson, and J. Vlissided. *Design Patterns*. Addison Wesley, 2008.

[39] Joseph C. Giarratano and Gary Riley. *Expert Systems*. PWS Publishing Co., Boston, MA, USA, 3rd edition, 1998.

[40] J. L. González. *Jugabilidad: Caracterización de la Experiencia del Jugador en Videojuegos*. PhD thesis, Universidad de Granada, 2010.

[41] T. Granollers. *MPIu+a. Una metodología que integra la ingeniería del software, la interacción persona-ordenador y la accesibilidad en el contexto de equipos de desarrollo multidisciplinares*. PhD thesis, Universitat de Lleida, 2004.

[42] J Gregory. *Game Engine Architecture*. AK Peters, 2009.

[43] Khronos Group. *COLLADA Format Specification*. http://www.khronos.org/collada/, 2012.

[44] H. Hoppe. Efficient implementation of progressive meshes. *Computers & Graphics*, 22(1):27–36, 1998.

[45] IberOgre. *Formato OGRE 3D*. http://osl2.uca.es/iberogre, 2012.

[46] R. Ierusalimschy. *Programming in LUA (2nd Edition)*. Lua.org, 2006.

[47] InfiniteCode. *Quadtree Demo with source code*. http://www.infinitecode.com/?view_post=23, 2002.

[48] Intel Corporation, Disponible en línea en `http://www.intel.com/content/dam/doc/manual/`. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3B: System Programming Guide, Part 2*, Mar. 2012.

[49] ISO/IEC. Working Draft, Standard for Programming Language C++. Document number N3242=11-0012., Feb. 2011.

[50] D. Johnson and J. Wiles. Effective affective user interface design in games. *Ergonomics*, 46(13-14):1332–1345, 2003.

[51] N.M. Josuttis. *The C++ standard library: a tutorial and handbook*. C++ programming languages. Addison-Wesley, 1999.

[52] G. Junker. *Pro OGRE 3D Programming*. Apress, 2006.

[53] G. Junker. *Pro Ogre 3D Programming*. Apress, 2006.

[54] C Keith. Agile game development tutorial. In *Game Developers Conference*, 2007.

[55] C Keith. *Agile Game Development with Scrum*. Addison-Wesley Professional, 2010.

[56] F. Kerger. *Ogre 3D 1.7 Beginner's Guide*. Packt Publishing, 2010.

[57] Donald E. Knuth. Structured Programming with *go to* Statements. *ACM Computing Surveys*, 6(4), Dec. 1974.

[58] V. Kurkova. Kolmogorov's theorem and multilayer neural networks. *Neural Networks*, 5(3):501–506, 1992.

[59] D. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 105–ff. ACM, 1995.

[60] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftmanship*. Prentice Hall, 2009.

[61] T. Masters. *Practical neural network recipes in C++*. Morgan Kaufmann, 1993.

[62] M. McShaffry. *Game coding complete*. "Paraglyph Publishing, 2003.

[63] S.D. Meyers. *Effective STL: 50 specific ways to improve your use of the standard template library*. Addison-Wesley professional computing series. Addison-Wesley, 2001.

[64] I. Millington and J. Funge. *Artificial Intelligence for Games*. Morgan Kaufmann, 2009.

[65] David R. Musser and Alexander A. Stepanov. Generic programming. In *Symbolic and Algebraic Computation: International symposium ISSAC 1988*, pages 13–25, 1988.

[66] J. Nielsen. *Usability Engineering*. AP Professional, 1993.

[67] NNG. *User Experience - Our Definition*. Nielsen Norman Group.

[68] Assembla NocturnaIR. *El Formato Collada*. http://www.assembla.com/wiki/show/reubencorp/ El_formato_COLLADA, 2012.

[69] ATI NVidia. Arb occlusion query, 2001.

[70] Manuel Palomo-Duarte, Juan Manuel Dodero, and Antonio García-Domínguez. Betting system for formative code review in educational competitions. *Expert Systems with Applications*, 41(5):2222–2230, 2014.

[71] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Revised Fourth Edition*. Morgan Kaufmann, 4th edition edition, 2012.

[72] E. Pipho. *Focus on 3D models*, volume 1. Course Technology, 2002.

[73] William T. Reeves. Particle systems - a technique for modelling a class of fuzzy objects. *ACM Transactions on Graphics*, 2:91–108, 1983.

[74] C.W. Reynolds. Steering behaviors for autonomous characters. In *Game Developers Conference*, volume 1999, pages 763–782, 1999.

[75] Gary Riley. CLIPS home site. http://clipsrules.sourceforge.net, 2013.

[76] R.J. Rost and J.M. Kessenich. *OpenGL Shading Language*. Addison-Wesley Professional, 2006.

[77] R. Rouse III. *Game Design: Theory and Practice*. Wordware Publishing, 2001.

[78] W.W. Royce. Managing the development of large software systems. In *Proceedings of IEEE WESCON*, 1970.

[79] S.J. Russell and Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 1998.

[80] E. Salen, K.; Zimmerman. *Rules of Play: Game Design Fundamentals*. The MIT Press, 2003.

[81] H. Schildt. *C++ from the Ground Up (3rd Edition)*. McGraw-Hill Osborne, 2003.

[82] D. Schmidt. Acceptor-connector: an object creational pattern for connecting and initializing communication services, 1997.

[83] B. Schwab and Inc ebrary. *AI game engine programming*. Charles River Media, 2004.

[84] P. Shirley and S. Marschner. *Fundamentals of Computer Graphics*. AK Peters, 3^{rd} edition, 2009.

[85] B. Shneiderman. Universal usability. *Communications of the ACM*, pages 84–91, 2000.

[86] D. Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1 (7th Edition)*. Addison-Wesley, 2009.

[87] B. Sierra Araujo. *Aprendizaje automático: conceptos básicos y avanzados: aspectos prácticos utilizando el software Weka*. Pearson Prentice Hall, Madrid, 2006.

[88] D. Sikora. Incremental game development.

[89] A. Silberschatz, P.B. Galvin, and G. Gagne. *Fundamentos de Sistemas Operativos*. McGraw Hill, 2006.

[90] Jouni Smed and Harri Hakonen. *Algorithms and Networking for Computer Games*. Wiley, 2006.

[91] R.M. Stallman and GCC Developer Community. *Using GCC: The GNU Compiler Collection Reference Manual*. GNU Press, 2003.

[92] R.M. Stallman, R. Pesch, and S. Shebs. *Debugging with GDB: The GNU Source-Level Debugger*. GNU Press, 2002.

[93] B. Stroustrup. *The C++ programming language*. Addison-Wesley, 2000.

[94] B. Stroustrup. *El lenguaje de programaci'on C++*. Pearson Education, 2001.

[95] I. Takeuchi, H.; Nonaka. Scrum: The new product development game, 1986.

[96] K. Tanaka. *An introduction to fuzzy logic for practical applications*. Springer Verlag, 1997.

[97] S.J. Teller and C.H. Séquin. Visibility preprocessing for interactive walkthroughs. In *ACM SIGGRAPH Computer Graphics*, volume 25, pages 61–70. ACM, 1991.

[98] T. Theoharis, G. Papaioannou, and N. Platis. *Graphics and Visualization: Principles & Algorithms*. AK Peters, 2008.

[99] José Tomás Tocino, Pablo Recio, and Manuel Palomo-Duarte. Gades Siege. `http://code.google.com/p/gsiege`, 2012.

[100] T. Ulrich. Rendering massive terrains using chunked level of detail control. *SIGGRAPH Course Notes*, 3(5), 2002.

[101] Wang and Niniane. Let there be clouds! *Game Developer Magazine*, 11:34–39, 2004.

[102] G. Weiss. *Multiagent Systems: a Modern Approach to Distributed Artificial Intelligence*. The MIT Press, 1999.

[103] Wikipedia. Stratego — wikipedia, the free encyclopedia, 2014. [Online; accessed 30-May-2014].

[104] M. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[105] L.A. Zadeh. Fuzzy sets. *Information and control*, 8(3):338–353, 1965.

[106] H. Zhang, D. Manocha, T. Hudson, and K.E. Hoff III. Visibility culling using hierarchical occlusion maps. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 77–88. ACM Press/Addison-Wesley Publishing Co., 1997.

Este manual fue maquetado en
una máquina GNU/Linux en
Junio de 2014